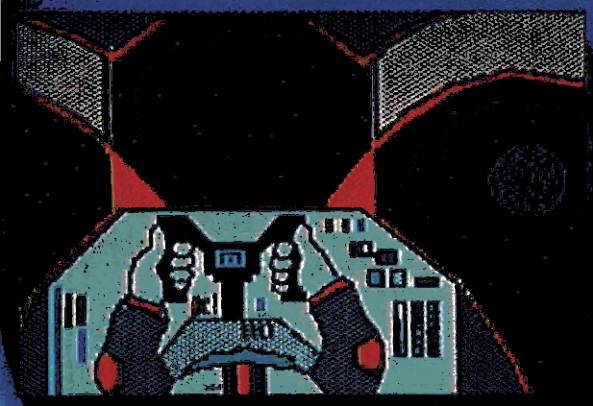
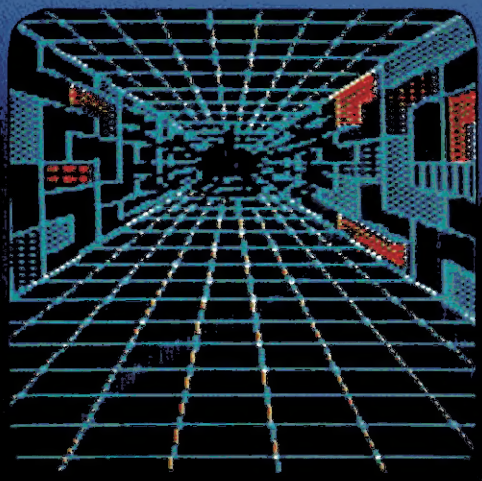
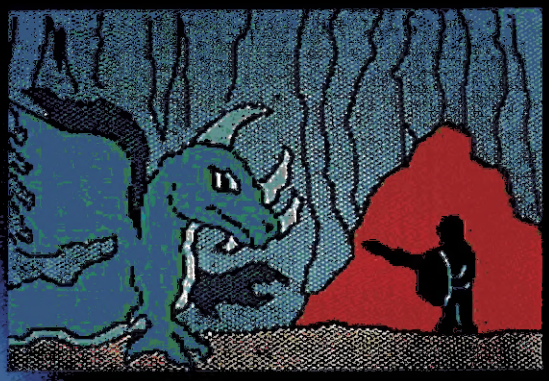
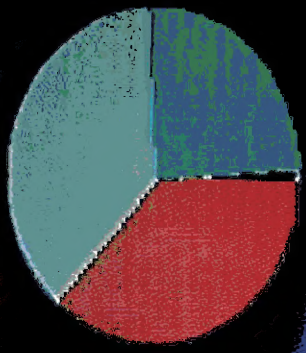
APPLE II COMPUTER GRAPHICS









Ken Williams • Bob Kernaghan • Lisa Kernaghan

APPLE II COMPUTER GRAPHICS

Ken Williams
Bob Kernaghan
Lisa Kernaghan

illustrations by
Gregory Paul Steffen

Robert J. Brady Co.
A Prentice-Hall Publishing and Communications Company
Bowie, MD 20715

Executive Editor: David T. Culverwell
Production Editor/Text Designer: Michael J. Rogers
Art Director: Bernard Vervin
Typesetting: Creative Communications Corporation
Cockeysville, MD
Typefaces: Eurostile (display); Optima (text)
Printed by: R.R. Donnelley & Sons Company
Harrisonburg, VA
Cover Design: Don Sellers
Indexer: Leah Kramer

Apple II Computer Graphics

Copyright © 1983 by Robert J. Brady Company.
All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Robert J. Brady Co., Bowie, Maryland 20715

Library of Congress Cataloging in Publication Data

Williams, Ken, 1938-
Apple II computer graphics.

Includes index.
1. Apple II (Computer)—Programming. 2. Computer graphics. I. Kernaghan, Bob. II. Kernaghan, Lisa. III. Steffen, Gregory Paul. IV. Title. V. Title: Apple 2 computer graphics. VI. Title: Apple Two computer graphics.
QA76.8.A662W54 1983 001.64'43 83-3871
ISBN 0-89303-315-4

Prentice-Hall International, Inc., London
Prentice-Hall Canada, Inc., Scarborough, Ontario
Prentice-Hall of Australia, Pty., Ltd., Sydney
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia Pte. Ltd., Singapore
Whitehall Books, Limited, Petone, New Zealand
Editora Prentice-Hall Do Brasil LTDA., Rio de Janeiro

Printed in the United States of America

83 84 85 86 87 88 89 90 91 92 93 10 9 8 7 6 5 4 3 2 1

Contents

1 Introduction **1**
2 Computer Physiology **3**
3 System Monitor—Memory Tricks **13**
4 APPLESOFT Extensions **21**
5 Graphics Modes and Soft Switches **31**
6 Text and Low-Res **39**
7 Preserving Your Pictures **53**
8 Hi-Res Graphics **65**
9 Hi-Res Color **81**
10 Shaping Up **99**
11 Graphs and Charts **119**
12 Byte-Move Shapes **135**
13 Advanced Moves **147**
14 Collision Course **157**
Appendix 1: Decimal, Hex, and Binary **165**
Appendix 2: Character Codes: ASCII vs APPLE **175**
Appendix 3: Memory Maps **177**
Glossary **181**
Index **186**

Foreword

The introduction of the Apple II computer marked a revolution in computers. For the first time there was a fully functional computing machine available which was priced affordably for home users. In addition to its powerful data processing capabilities, APPLESOFT had the first set of BASIC extensions which allowed the programmer true power and convenience when working with graphics. The six-color high resolution mode with the HCOLOR and HPLOT commands were "state of the art," and the SHAPE construct and associated DRAW, XDRAW, SCALE, and ROTate functions were ahead of their time.

When I acquired my first Apple, the Low-Res "Little Brick Out" was considered a very good game, and Bob Bishop's "Applevision" demo was, and still is, awe-inspiring. But with literally millions of people seeing and using the Apple, advances in style and technique came rapidly as programmers and "hackers" accepted and surmounted one challenge after another in their attempt to "do it better."

The manuals state that only six colors are available on the High-Res screen, but Sierra On-Line soon developed over 100 colors. It was not supposed to be possible to put text on the High-Res screen, but doing so is now commonplace. The power, ingenuity, and persistence of the human mind has extended the capabilities of the Apple further than anyone had imagined possible. Like most of the current software companies, Sierra On-Line was born in long and solitary hours of developing newer, faster, and better methods of producing graphics on the Apple. In writing this book, it is my hope to introduce you to those advanced techniques such as Byte-Move, animation, and collision detection, as well as the standard Apple graphics.

With the aid of my co-authors, who are professional teachers in addition to being experts in the field of home computers, I have tried to present those ideas in a manner which is clear and understandable, and which relies only on a knowledge of BASIC. I hope you find this book enjoyable to read as well as useful in your own program development.

Ken Williams

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

1

Introduction

The level of graphics for the Apple II has risen considerably since the early days when Little Brick Out was considered a major accomplishment. The industry standards now virtually require that good graphics be multi-colored and high resolution, and that the animation be smooth, clean, and fast, and collision detection quick and accurate.

If you are at all familiar with the remarkable graphics effects on the market today, you have probably become curious about the methods used to generate them. This text will detail those aspects of Apple II graphics which are currently in common use; particularly with regard to Hi-Res, Low-Res, and Text Mode graphics, color generation, animation, and collision detection. All subjects are presented for use by the APPLESOFT programmer, but have extensions for the assembly and machine language programmers.

In using this text, it is necessary to have the following equipment handy and ready to use:

Apple II, Apple II+, or Apple IIe computer

Color Monitor or TV

One or more disk drives and disks

or

A tape recorder and tapes

Lots of paper

A printer is helpful for graphics, but not necessary.

This book is not written for the beginning BASIC programmer; it is necessary that the user of this book have some knowledge of BASIC program-

ming. The user should be familiar with the following commands: GR, COLOR, PLOT, HLIN, VLIN, SCRN, HGR, HGR2, HCOLOR, HPLOT, DRAW, and XDRAW. If you are unfamiliar with APPLESOFT BASIC or these commands, please review the *APPLESOFT Reference Manual* before continuing.

Before starting our journey into the world of graphics, it is prudent to arm ourselves with some relatively technical, often frustrating, but quite essential material. We assume you have a rudimentary knowledge of APPLESOFT BASIC and your Apple II system. So, let us begin with a discussion of the binary, decimal, and hexadecimal systems of numeration.

The time has come, the walrus said
To speak of many things.
Of bits and bytes and peeks and pokes . . .

2

Computer Physiology

Objectives

After reading Chapter 2 you should be able to:

- Convert binary numbers to hexadecimal numbers and back.
- Convert hexadecimal to decimal.
- Understand the idea of memory addressing.
- Use the post office box model for computer memory.

Most beginning programmers work exclusively with the software phases of computing, and are content to leave the concern for such things as circuitry, memory allocation, and video display to computer engineers. But when one tries to work with microcomputer graphics using only the constructs and commands available to BASIC or other high-level languages, he or she finds the going slow and awkward. To effectively utilize the graphics capabilities of your machine, you need at least a rudimentary conceptual understanding of what is happening within the beast.

To gain that necessary understanding, you will need a working knowledge of the binary, decimal, and hexadecimal systems of numeration. **Decimal** is the common base ten system which you ordinarily use, **binary** is the base two system which your computer uses, and **hexadecimal,** or **hex,** is the base sixteen system which is a useful compromise serving as the system of communication between you and your computer.

Although your computer is capable of "speaking" BASIC and perhaps other languages, good graphics effects often require you to deal directly with system memory and use some machine level routines. Working at the

machine's level means learning something about how the computer stores and processes data.

Digressing to Digital

All of today's microcomputers use two-state digital circuitry. In relation to computers, digital circuitry means, in very loose terms, that all information is represented within the computer as a discreet state of a circuit—usually either off or on. The power light on your Apple is a good example; the light is either off or on, and by looking at the light, you can infer the state of your machine.

There are analog computers whose basis for processing is not the presence or absence of voltage, but instead the amount of voltage present in a circuit. However, it is operationally much easier to test for the presence or absence of voltage in a circuit than it is to determine the precise amount of voltage. You can imagine the complexity and accuracy required of a computer circuit to correctly and reliably represent a number such as 36,741 by analyzing only the amount of the voltage. For that reason the memory of your computer is digital, and consists of thousands of two-state microcircuits, where each circuit can be set either on or off. Since each circuit maintains or "remembers" the on or off state to which it was set, these circuits can be used to store all of the information which the machine is supposed to remember.

RAM and ROM

Your Apple II has two types of memory: RAM and ROM. **ROM** stands for **R**ead **O**nly **M**emory. The information contained in ROM is essentially permanent and unalterable. You may read the information from ROM, but you cannot write to it or change it in any way (that is why it is termed "Read Only"). Your version of BASIC is stored in ROM, as are most of the other instructions which make your system operate. That is what makes the Apple II "intelligent" since its operating system is "on board" or "ROM resident."

RAM is short for **R**andom **A**ccess **M**emory; RAM is the working memory for your Apple. Whether you are writing, editing, or running a program, that program resides in RAM, as do all of the variables and other data you use. RAM is of great interest in graphics because the video output is simply displaying data from a secton of the RAM. The use of graphics then becomes mostly a matter of putting the correct data into the proper place in memory.

To make it easier to visualize, memory within the computer may be thought of as a great many switches, each of which is either on or off at any given time. Those memory switches are commonly called "bits," and

ONE BIT

ON
OFF

TWO CHOICES

2^1

TWO BITS

FIRST BIT	SECOND BIT
ON	ON
ON	OFF
OFF	ON
OFF	OFF

FOUR CHOICES

2^2

THREE BITS

FIRST BIT	SECOND BIT	THIRD BIT
ON	ON	ON
ON	ON	OFF
ON	OFF	ON
ON	OFF	OFF
OFF	ON	ON
OFF	ON	OFF
OFF	OFF	ON
OFF	OFF	OFF

EIGHT CHOICES

2^3

Figure 2-1. Bit combinations.

once each RAM bit is set on or off, it remains in that state until altered by a command, or until the computer is turned off. (At which time the values in RAM are lost.) All of the computer's impressive capabilities rely on the lowly bit. One bit is quite a long way from making a computer, just as one grain of sand is a long way from being a beach. A bit, like a grain of sand, has little meaning individually, but when grouped with thousands of other bits, together they are able to take on meaning and form.

Making It Happen with Ones and Zeros

As stated earlier, each bit will take on only one of two states, on or off. In order to facilitate discussion, if a bit is on, it is said to have a value of 1; a bit which is off has a value of Ø. Since each bit may only be 1 or Ø, each bit conveys a very limited amount of information. In the example of your

computer's power light, you know only that the power is on or off, and nothing more. Two bits in a group combine in one of four (2^2) combinations: 00, 01, 10, and 11. (10 and 11 are interpreted as one-zero and one-one, respectively, not as ten and eleven.) Using the computer's power light together with the "in use" light of the disk drive can illustrate how two digital circuits can convey four pieces of information: If both lights are off, neither unit is operating. If the power light is on and the disk light is off, the computer is on but not the disk drive. If the power light is off but the disk light is on, the disk is operating but not the computer!?!!? (time to call the repairman). If both lights happen to be on, then both units are operating. Similarly, a group of three bits allows for eight (2^3) combinations, and a group of four bits allows for sixteen (2^4). The number of possible combinations continues to increase exponentially as does the number of bits used. Figure 2–1 illustrates the possible combinations for one, two, and three bits.

Bits in a Bag

Your computer contains hundreds of thousands of bits. As we have seen, however, they must be grouped together in order to convey any significant information. Most microcomputers, including your Apple, group eight bits into a unit called a "byte." A **byte** is the fundamental unit of memory. The size of memory is usually discussed in terms of kilobytes ("K" for 1,000), or megabytes ("M" or Meg, for 1,000,000).

A single eight-bit byte can represent any one of 256 possible values (2^8 = 256). Having 256 combinations works out conveniently to allow any letter or digit be assigned to one of those 256 combinations and represented by it; the computer stores the character by storing the numeric code. If bits were only grouped six at a time, for example, you would only be able to form 64 (2^6) distinct combinations. However, the computer needs to have more than 64 distinct bit patterns available in order to uniquely represent each of the required characters, including all the keys on the computer keyboard—both upper and lower case, the special characters such as & and *, and the control codes such as CTRL-C.

When you press a key, the computer actually receives an eight bit pattern; when you press "A" your Apple sees the pattern 11000001, and when you press the "*" key, the computer sees 10101010. Each character has its own unique bit pattern. After you press a key, the computer stores that particular bit pattern in a byte of memory. But now that the information is stored, the computer has to be able to locate it when it is needed, which leads us to....

Addressing

Each byte in the Apple's memory is numbered sequentially, and that num-

ber is called the "address" of that byte. A memory address is much like the number on a post office box—one can determine what is in any byte of memory by going to the box with the proper address and looking inside. The number of boxes available depends upon two things: The number of boxes installed (memory size), and the maximum number of digits available in the address. If you were limited to use only three decimal digits in an address, there would be a maximum of 10^3 (1000) boxes—addressed 000 through 999. The processor within the Apple allows for two bytes (sixteen bits) in the address of any location in memory, and that provides a maximum of 2^{16} addresses. If you have wondered why you can use a maximum of 64K bytes of memory within an Apple II, consider that each memory location has to have an address, and that 2^{16}, using a little arithmetic and a good calculator, equals 65,536—the true maximum for the number of bytes the 6502 processor can address. Many larger computers use three or more bytes for addressing; a three-byte address contains 24 bits, so by allowing three bytes in the addresses, a computer can address 2^{24} memory locations: over SIXTEEN MILLION bytes of memory!

Binary Blindness

Your computer is very happy to work in binary. The early machines did so exclusively, but most people find it trying—looking at binary numbers all day makes your eyes cross! To demonstrate, compare the two columns of binary numbers below, and try to determine where they differ.

10010111	10010111
01010110	01010110
00010010	00010010
11011010	11011010
00100111	00100111
11101100	11100100
01000001	01000001
11110100	11110100

Is it any wonder that there were very few programmers in the days when computers spoke only binary?

Hex to the Rescue

The hexadecimal or hex system of numeration, base sixteen, supplies a welcome solution to the problem of "binary blindness." Hex is advantageous because it provides a convenient way to turn the binary ones and

zeros into something more easily communicated to the human brain, so most modern computers like to speak to people using hex.

Hexadecimal, which uses sixteen digits, is used to represent the 256 different combinations available with an eight-bit byte. It is conceivable to represent the value of a byte using a single base 256 digit; however, most people would find it very difficult to remember 256 different digits. Therefore, in the interest of simplicity, instead of one base 256 digit representing the value of a byte, two hex digits are used. Why not use decimal which is simpler yet? Do not forget that we are dealing with a group of bits. The ten decimal digits are too many to represent three bits and too few for four bits, as three bits make for eight combinations, and four bits make for sixteen. The fact that sixteen combinations are available to four bits, and that hexadecimal uses exactly sixteen digits, is very important. This is the reason that many programmers count like they have sixteen fingers!

BIT 1	BIT 2	BIT 3	BIT 4	HEXADECIMAL	BINARY	DECIMAL
OFF	OFF	OFF	OFF	0	0 0 0 0	0
OFF	OFF	OFF	ON	1	0 0 0 1	1
OFF	OFF	ON	OFF	2	0 0 1 0	2
OFF	OFF	ON	ON	3	0 0 1 1	3
OFF	ON	OFF	OFF	4	0 1 0 0	4
OFF	ON	OFF	ON	5	0 1 0 1	5
OFF	ON	ON	OFF	6	0 1 1 0	6
OFF	ON	ON	ON	7	0 1 1 1	7
ON	OFF	OFF	OFF	8	1 0 0 0	8
ON	OFF	OFF	ON	9	1 0 0 1	9
ON	OFF	ON	OFF	A	1 0 1 0	10
ON	OFF	ON	ON	B	1 0 1 1	11
ON	ON	OFF	OFF	C	1 1 0 0	12
ON	ON	OFF	ON	D	1 1 0 1	13
ON	ON	ON	OFF	E	1 1 1 0	14
ON	ON	ON	ON	F	1 1 1 1	15

Figure 2-2. A binary/hex chart.

Taking a Smaller Byte

The next problem is to represent each of the 256 different values for one byte using only the sixteen hex digits—the standard Ø through 9, plus the letters A, B, C, D, E, and F; where A stands for ten, B for eleven, and so forth through F which stands for fifteen.

One byte is represented by two hex digits; each digit represents four of the eight bits in that byte. As modern computer whimsy would dictate, each of the two half-bytes is called a nibble (sometimes nybble). A binary nibble corresponds to a hex digit as shown in Figure 2-2.

Suppose that you were looking at a byte with all eight bits turned on. Separated into the two nibbles, it would look like 1111 1111 and would correspond to hexadecimal FF, since each nibble (1111) corresponds to F. The binary byte 1Ø1Ø ØØØØ is hex AØ, and binary Ø111 11ØØ is hex 7C. (See Figure 2-3.)

Trying to remember which binary number goes with which hex number can make your head hurt, so you might want to make your own chart like the one in Figure 2-2 and leave it in a handy place as an aid in converting between hex and binary.

Since it is not always clear whether a numeral such as 1Ø is in hex, decimal, or even binary, it is conventional to write a hexadecimal numeral with a dollar sign ($) in front, such as $FF. Binary numerals are preceded by exclamation points (!), and decimal numerals are left alone.

Conversion to Decimal

When dealing directly with the memory of your machine, it is usually best not to convert to decimal. Thinking in hex is often more efficient since it is

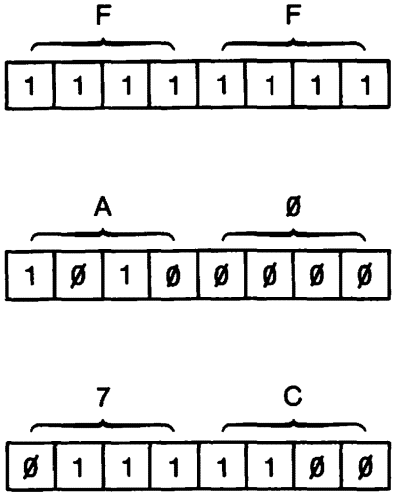


Figure 2-3. Binary/hex examples.

the system your Apple understands. There are, however, a few instances where you must convert hex to decimal, such as for use in an APPLESOFT POKE or CALL statement. When you do have to perform the conversion you may use the following algorithm:

Reading the hex number from right to left, the decimal value of the entire number is found by taking the decimal value of the first hex digit (nibble), plus 16 times the decimal value of the second digit, plus 256 (16^2) times the decimal value of the third digit, plus 4096 (16^3) times the decimal value of the fourth digit, and so on.

Example 1

$$\begin{aligned}\$C057 &= 7 + 16*5 + 256*0 + 4096*11\\ &= 7 + 80 + 0 + 45056\\ &= 45143\end{aligned}$$

Example 2

$$\begin{aligned}\$10 &= 0 + 1*16\\ &= 16\end{aligned}$$

Microcomputers rarely deal internally with numbers having more than four hex digits, but the algorithm does carry on if necessary. If you convert the largest possible two byte hex number, $FFFF, to decimal you should get 65,536. Since your micro uses two byte addresses, you can again see why that number is the largest in the Apple's world.

Although conversion methods have been discussed only briefly, the preceding discussion is sufficient to serve our purposes for now. Appendix 1 contains a more thorough treatment of hexadecimal, decimal, and binary number systems. There are also a number of calculators presently on the market which are designed for computer programmers; they have functions to do all of the conversions for you at the press of a button.

Vocabulary

Address	Hexadecimal
Binary	Memory
Bit	Nibble
Byte	RAM
Decimal	ROM
Hex	ROM Resident

Exercises

1. Convert the following binary numbers to hex:
   a. !1Ø11 Ø1ØØ
   b. !Ø1Ø1 1111
   c. !Ø1Ø1 Ø11Ø
2. Convert the following to binary:
   a. $7C
   b. $FF
   c. $A5
   d. $DB
3. Convert the following to decimal:
   a. $7C
   b. $FF
   c. $A5
   d. $DB
   e. $7CFF
   f. $DBA5
   g. !1Ø11 ØØØØ (Hint: convert to
   h. !Ø111 Ø1Ø1 hex first.)
4. Convert $3F and $57 to binary and to decimal numerals.
5. Convert !1ØØ1 Ø11Ø and !Ø111 1Ø11 to hexadecimal.

3

System Monitor— Memory Tricks

Objectives

After reading Chapter 3 you should be able to:

- Enter and leave the Monitor.
- Examine RAM and ROM.
- Change the contents of RAM.
- Move blocks of memory.
- Use a memory map to locate reserved and free memory.

In essence, your computer is simply a pile of very fancy circuitry, and those circuits are designed to respond to a certain set of very, very rudimentary instructions. Although BASIC will respond to high level commands such as NEW and PRINT, those are commands of your language, and not your processor. It is the function of the BASIC interpreter to translate those statements down to the elementary ones and zeros that your processor requires to operate; the machine takes over from there. The capabilities of BASIC are good as far as they go, but there are applications to graphics which require that the programmer deal directly with memory.

Memory is where everything happens, and though you can reach memory using BASIC's PEEK and POKE statements, there is a faster, more powerful, and sometimes more convenient method. Supplied in the ROM of every Apple is the system Monitor, with a capital M, which is very

different from your video monitor. The Monitor is actually another computer language and is designed to let you, the user, communicate directly with system memory. As related to graphics, you will use Monitor to inspect, modify, or move the contents of the RAM. You will also learn to use Monitor to enter short machine language programs which can be accessed from your BASIC programs to manipulate graphics. But before you can learn about the Monitor, you have to find it.

Monitor, I Presume

If you have worked much with your Apple you have noticed that, when you are talking to APPLESOFT, the computer displays a] on the left edge of the screen; you get a > when you work with INTEGER BASIC. Those are prompt characters, and their purpose is to remind you which language you are dealing with. The prompt character for Monitor is *. If you have a regular Apple II, you see the * prompt every time you turn the system on. When you power up an Apple II+ or Apple IIe, they will immediately access the disk and load the disk operating system (DOS). After DOS has been loaded, the screen will display either the > or] prompt, and then you can reach the Monitor by typing:

```
CALL-151 <CR>
```

When you want to return to BASIC from Monitor, type:

```
<CTRL-C><CR>
```

If you have DOS up, an alternate method of returning to BASIC is to type:

```
3DØG<CR>
```

Examining Memory

You can use Monitor to examine or alter any byte of RAM; you can also examine ROM, though you cannot change it. Enter the Monitor, then type:

```
33<CR>
```

The display should respond with the contents of address $33:

```
ØØ33- AA
```

Location $33 is where Apple stores the current prompt character, and $AA is Apple's numeric code for the *. Notice that you did not preface the 33 with a dollar sign, even though you meant hexadecimal 33; Monitor always speaks in hex and assumes that you will do the same.

You may look at more than one byte of memory at a time by specifying the beginning and ending addresses of the range of memory you wish to see. Still, from Monitor type:

```
CØØ.C1F<CR>
```

Do not use spaces to separate the two elements of the command. The computer will then display something similar to:

```
ØCØØ- xx xx xx xx xx xx xx xx
ØCØ8- xx xx xx xx xx xx xx xx
ØC1Ø- xx xx xx xx xx xx xx xx
ØC18- xx xx xx xx xx xx xx xx
```

The numbers in the left hand column index the memory locations displayed in the first column of each line. Each of the xx's will be some hexadecimal byte, though the precise value of each byte will vary. Shown in the first row will be the contents of memory locations CØØ, CØ1, CØ2, CØ3, CØ4, CØ5, CØ6, and CØ7. The second row gives the contents of CØ8, CØ9, CØA (remember to count in hex), CØB, CØC, CØD, CØE, and CØF. The third row begins at C1Ø and continues in a similar manner.

Changing Your Memory

You can use Monitor to alter the value in any given byte of RAM. To change the eight bytes beginning with the byte at CØØ, enter:

```
CØØ:FF FF FF FF FF FF FF FF<CR>
```

The colon is the Monitor command to change the values of memory starting at the given location (CØØ), and the FF's are the values to which you wish to change. The spaces between each of the bytes are necessary for the command to execute properly. (From here on out, we will assume that you will press the return key when required, so we will stop putting <CR> after each entry.)

To verify that the memory has been changed, enter:

```
CØØ.CØ7
```

and this time you will see:

```
ØCØØ- FF FF FF FF FF FF FF FF
```

This indicates that the eight bytes beginning at CØØ all contain $FF.

Memory Organization

Like all microcomputers, the Apple's RAM is not all available for your use; some portions are used by the DOS, some by the Monitor, and some areas are reserved for temporary storage (buffers) or "scratch pad" areas. In the previous example, we chose to alter the range CØØ through CØ7 because that area is not used by the computer for its own purposes, so it is relatively safe. If you alter values in an area used by the system, such as the

DOS vector area ($3CØ–$3FF), you have an excellent chance of creating havoc, so be very careful. The memory map in Figure 3–1 shows which areas of memory are used for what.

Memory Map of a 48K Apple II

Function	Address
APPLE MONITOR	$F800-$FFFF
APPLESOFT	$E000-$F7FF
RESERVED	$D000-$DFFF
I/O DECODE	$C000-$CFFF
DOS	$9600-$BFFF
UNUSED	$6000-$95FF
HI-RES PAGE 2	$4000-$5FFF
HI-RES PAGE 1	$2000-$3FFF
UNUSED	$C00-$1FFF
TEXT/LO-RES PAGE 2	$800-$BFF
TEXT/LO-RES PAGE 1	$400-$7FF
DOS VECTORS	$3C0-$3FF
UNUSED	$300-$3BF
TEXT INPUT BUFFER	$200-$2FF
6502 STACK	$100-$1FF
ZERO PAGE	$0-$FF

Figure 3–1. A memory map.

In a 48K system, RAM extends from location $Ø, at the "bottom" of memory, through location $BFFF. The memory above $BFFF ($CØØØ through $FFFF) is ROM. (Unless you own a 16K RAM expansion, in which case the entire range is RAM.) The reference manuals for your system contain several memory maps which are useful in determining which areas are used by the system, BASIC, and/or DOS.

Write From Memory

You can place characters directly on the Text Screen by inserting the numeric code for the character into the area of memory which is displayed (Text, page 1, is the area of memory displayed). The memory map shows that the memory area for page 1 of Text lies between $4ØØ and $7FF. To clear the screen from Monitor, type:

```
<ESC><SHIFT-P>
```

Locations $6DØ through $6D7 are within Text Screen Memory, so let's place the Apple code for "B" in those locations by typing:

```
6DØ: C2 C2 C2 C2 C2 C2 C2 C2
```

As soon as you press the return key, eight "B's" will appear near the bottom of your screen.

Next, change the same locations to $AØ, and the "B's" will disappear. As you have probably guessed, $C2 is the Apple code for "B", and $AØ is the code for a blank. You could clear the entire screen by placing $AØ in every location from $4ØØ through $7FF, but entering 96Ø copies of $AØ is not only tedious, it can be avoided by using the Monitor MOVE command which we will discuss next.

Making Memory Move

The Monitor MOVE command lets you move the contents of one range of memory to a different range. For example, you could use the MOVE command to take the contents of the eight bytes which begin at location $CØØ and place those values into the eight bytes which begin at $DØØ. You would have to tell Monitor several things: that you wish to move memory, which chunk of memory is to be moved (CØØ.CØ7), and where you wish to put the values that you are moving (DØØ).

First, display the contents of $CØØ through $CØ7 just to see what is there. Enter:

```
CØØ.CØ7
```

then move those values to $DØØ by typing:

```
DØØ<CØØ.CØ7M
```

Do not use spaces to separate the different parts of the command. This command tells Monitor that the memory locations beginning at $DØØ are to receive (<) the contents of memory from the range $CØØ.CØ7. The "M" at the end stands for move. Reverting to the analogy of memory being similar to post office boxes, the MOVE command is similar to taking the mail in the eight boxes numbered $CØØ through $CØ7, and shifting the mail from each box up $1ØØ boxes so it ends up in the boxes numbered $DØØ through $DØ7 as illustrated in Figure 3-2.

STEP #	CONTENTS OF		SHIFTED TO
1	$CØØ	---->	$DØØ
2	$CØ1	---->	$DØ1
3	$CØ2	---->	$DØ2
4	$CØ3	---->	$DØ3
5	$CØ4	---->	$DØ4
6	$CØ5	---->	$DØ5
7	$CØ6	---->	$DØ6
8	$CØ7	---->	$DØ7

Figure 3-2.

The post office box model does have a weakness; the information in the original locations is not actually taken out of the old boxes and moved to the new locations, but rather a copy is made and then stored at the new addresses. Hence the same values actually reside in both memory ranges, and the values which were originally in the destination addresses are destroyed when using the MOVE command.

Now it is time to learn to do something quasi-useful with the MOVE command: clear the text screen (memory locations $400-$7FF). Begin by placing the Apple code for a blank, $A0, into $3FF, by typing:

```
3FF: A0
```

Next enter

```
400<3FF.7FEM
```

and the entire screen will clear. You executed a MOVE command where the source range of memory ($3FF.7FE) overlapped with the destination range ($400.7FF). This can produce some startling results and warrants a closer look.

The Monitor MOVE command works on one byte at a time and proceeds in ascending order. Therefore, the first step copied the contents of $3FF (an A0) into $400, the second step copied the contents of $400 (now also A0) into $401, and so on as shown in Figure 3-3.

STEP #	CONTENTS OF		SHIFTED TO
1	$3FF	---->	$400
2	$400	---->	$401
3	$401	---->	$402
4	$402	---->	$403
5	$403	---->	$404
.	.		.
.	.		.
.	.		.
958	$7FC	---->	$7FD
959	$7FD	---->	$7FE
960	$7FE	---->	$7FF

Figure 3-3.

Though this process may seem a bit complex at first blush, with a little practice it becomes both routine and useful.

The memory map in Figure 3-1 shows that location $3FF is within an area of memory used by DOS. To avoid any possibility of clobbering DOS, it is a good idea to replace the value that was originally there, so use what you have learned so far to put an $FF back in address $3FF.

Other Monitor Tricks

There are other Monitor commands, such as the command you use when you return to BASIC by typing 3DØG. The 3DØ is the address marking the beginning of a DOS routine, and the G says GO. However, the value of those other commands to the BASIC-oriented graphics programmer is marginal, so we will not discuss them further here. If you are interested, refer to Chapter 3 of the *APPLE II Reference Manual* for more detail.

Vocabulary

Buffer

Memory Map

Monitor

MOVE

Prompt Character

Exercises

1. Use the Monitor to examine the contents of locations $96ØØ through $96FØ.
2. Examine memory locations $AA6Ø and $AA61 using a single Monitor command.
3. Examine location $A4 without displaying the contents of any other location.
4. Load $FF's into locations $CØ5 and $CØ6.
5. Move the contents of $4ØØ through $7FF to the block of memory which begins at $61ØØ.
6. Load $ØØ into each location from $2ØØØ through $3FFF.

4

APPLESOFT Extensions

Objectives

After reading Chapter 4 you should be able to:

- Use the PEEK statement to determine the value in any given memory location.
- Use POKE to place given values into given memory locations.
- Use CALL and USR to execute a given machine level routine in memory.

APPLESOFT BASIC contains a number of additions or extensions to the standard set of BASIC commands. Among those extensions are several commands relating to graphics, such as GR, HGR, HLIN, and PLOT. Although these commands embody the ease and simplicity of the BASIC language, they also share its drawbacks—slow execution and limited access to system memory. There is another set of command extensions, the PEEK, POKE, CALL, and USR statements, which allow the APPLESOFT programmer a good deal of access to memory and machine-level operations; this in turn permits more control and faster execution of graphics effects.

Take a Peek

The PEEK command does very closely what its name implies; it PEEKs at any given memory location to see what value is stored there. PEEK interro-

gates memory from within a BASIC program, without having to enter the Monitor. The command PRINT PEEK (37), for example, will print the contents of the thirty-seventh location in memory. PEEK is a decimal-oriented command, so in the previous example both the address (37), and the value printed are in decimal form.

Another application of the statement involves PEEKing to see if a key has been pressed. This is useful when the computer is required to process continuously and still look for a signal to interrupt, such as a change of data. Many of the computer-based arcade games provide examples of this since they must continuously animate the figures on the screen, but still check for keyboard, game paddle, and/or joystick input in order to receive the signal to fire, change the player's direction, and so on.

To illustrate use of the PEEK statement, key in the program given in Listing 4-1. When you have finished keying in the program, run it. What do you see? You should see the letter "L" being continuously printed on the screen. Press another key. The character which was pressed will now be printed until yet another key is pressed, and so on. Program execution may be stopped by pressing the reset key.

```
10 REM DEMONSTRATE PEEK
20 REM
30 P$ = "L"
40 REM
50 REM TOP OF LOOP
60 PRINT P$;
70 P = PEEK (49152): REM SEE IF KEY PRESSED
80 IF P > 127 THEN PRINT CHR$(7);:
   P$ = CHR$(P-128): POKE -16368,0
90 GOTO 50
```

Listing 4-1.

The following line-by-line explanation should clarify the example and its use of the PEEK command.

```
30 P$ = "L"
```

Line 30 assigns P$ an initial value of L.

```
60 PRINT P$;
```

Line 60 prints the value of P$. The semicolon suppresses the usually automatic line feed so that the printing will continue across the screen.

```
70 P = PEEK (49152): REM SEE IF KEY PRESSED
```

Line 70 assigns P the value in location 49152. Location 49152 has two functions: the first is to signal when a key has been pressed, and the second is to temporarily hold the numeric code for that key. When a key is pressed, the left-most bit in byte 49152 is turned on, so the byte has a

value of 128 (or greater if any other bit in byte 49152 happens to be on). Also, the value of the key pressed is stored in the other seven bits of byte 49152. If the value in location 49152 is greater than 127, then a key has been pressed.

```
8Ø IF P > 127 THEN PRINT CHR$(7);:
   P$ = CHR$(P-128): POKE 49168,Ø
```

If a key is pressed, then printing CHR$(7) will sound the bell, and P$ is reassigned to be the character pressed. CHR$(P-128) translates the numeric code for the character into the actual character. The POKE readies the keyboard for the next input by changing the value of location 49152 to a value less than 128.

```
9Ø GOTO 5Ø
```

This line begins the loop again.

POKE'n Along

The POKE command also states its own function. It lets you POKE a value into memory from within BASIC, without using the Monitor. The effect of POKE 37,12 is to place the value 12 into memory location 37. As with the PEEK statement, the arguments (37 and 12) are decimal-form numerals. As it happens, the contents of location 37 determine the vertical position of the cursor, so POKE 37,12 is equivalent to VTAB (12).

The program given in Listing 4–2 demonstrates a use for POKE which has direct application to graphics. Type in Listing 4–2 and run it. You will see the Hi-Res screen fill with dots and lines, all of this occurring without the use of the HPLOT or HLINE commands. The program POKEs random values directly into locations within the Hi-Res screen memory area, so that the dots corresponding to the "on" bits of the byte will appear on the Hi-Res display. You will have to press reset to stop the program. In later chapters you will calculate some special values, and create pictures on the Hi-Res screen using this method.

```
1Ø REM DEMONSTRATE POKE
2Ø REM
3Ø HGR
4Ø REM
5Ø REM TOP OF LOOP
6Ø L% = (RND(1) * 8192) + 8192
7Ø V% = (RND(1) * 256)
8Ø POKE L%,V%
9Ø GOTO 5Ø
```

Listing 4–2.

The following line-by-line explanation should clarify the example.

```
3Ø HGR
```

sets the Hi-Res graphics mode and clears the screen.

```
6Ø L% = (RND(1) * 8192) + 8192
```

generates a random location between 8192 ($2000) and 16383 ($3FFF), inclusive. This area of memory is reserved for page 1 of Hi-Res graphics.

```
7Ø V% = (RND(1) * 256)
```

generates a value between Ø and 255, inclusive.

```
8Ø POKE L%,V%
```

POKEs the value into the memory location within screen memory to make the dots appear.

```
9Ø GOTO 5Ø
```

begins the loop again.

Call for Help

The CALL statement is much like the APPLESOFT GOSUB, except that the subroutine being CALLed is a machine language routine in memory instead of a BASIC routine within your program. At the end of the subroutine, the machine code equivalent of a RETURN statement returns to the instruction which followed the CALL in your program. The use of machine level routines is helpful because they allow you more control over the computer, and are much faster than corresponding BASIC routines. Smooth animation virtually requires the extra speed afforded by machine level subroutines. On the negative side, the workings of a routine written in machine code are usually rather obscure; and further, any errors in using that routine can lead to any number of unpredictable and bizarre results.

The simplest use of the CALL statement is to CALL one of the routines within the Apple operating system. CALL -936 clears the text screen and moves the cursor to the upper left corner—the same effect as the HOME statement. When one uses the HOME command, APPLESOFT essentially issues a CALL to that same location. There are many other usable routines listed in the APPLESOFT manual, see pages 129-13Ø, and the APPLE II REFERENCE MANUAL, see pages 61–64.

Another way to use CALL is to have your BASIC program POKE a machine language program into memory, then invoke that routine using the CALL statement. Type in the program from Listing 4–3 by way of an example. You will see the screen will fill with the inverse @ character. The first portion of the program POKEs a subroutine into memory using a data table, and then the subroutine is CALLed. The subroutine itself is of no real value because all it does is fill the screen with inverse @ characters, but it does demonstrate the use of CALL.

```
10 REM DEMONSTRATE CALL
20 REM
30 REM POKE THE MACHINE
40 REM LEVEL ROUTINE
50 REM INTO MEMORY
60 REM
70 FOR L = 768 TO 802
80 READ V
90 POKE L,V
100 NEXT L
110 REM
120 REM POKES COMPLETE
130 REM
140 PRINT "GET READY"
150 FOR I = 1 TO 1000: NEXT I
160 CALL 768
170 VTAB (24): PRINT "ALL DONE":
    GOTO 250
180 REM
190 REM DATA TO POKE
200 REM
210 DATA 169,0,133,254,169,7,13
    3,255,160,255
220 DATA 162,4,32,88,252,165,16
    1,145,254
230 DATA 198, 254,208,252,136,2
    08,247,202,240,5,198,255,76
240 DATA 17,3,96
250 END
```

Listing 4-3.

To aid your understanding, each line is explained below.

```
70 FOR L = 768 TO 802
80 READ V
90 POKE L,V
100 NEXT L
```

This loop reads the values from the data table and POKEs them into memory locations 768 through 802. This is a common method of placing a machine level routine into memory.

```
140 PRINT "GET READY"
150 FOR I = 1 TO 1000: NEXT I
```

These lines build the suspense to great heights!

```
160 CALL 768
```

This line executes the subroutine which was set up by lines 70–100.

```
17Ø VTAB (24): PRINT "ALL DONE":
    GOTO 25Ø
```

After the subroutine is complete, this line will print an end of job message and then jump to the END statement.

```
21Ø DATA 169,Ø,133,254,169,7,13
    3,255,16Ø,255
22Ø DATA 162,4,32,88,252,165,16
    1,145,254
23Ø DATA 198, 254,2Ø8,252,136,2
    Ø8,247,2Ø2,24Ø,5,198,255,76
24Ø DATA 17,3,96
```

Lines 21Ø-24Ø constitute the data table of values to be placed into memory to form the subroutine. They are READ by line 8Ø and POKEd by line 9Ø.

Using the technique presented above, any machine language program may be POKEd into memory, and then executed from BASIC using the CALL statement.

USR

The USR statement is similar to the CALL statement, in that it causes the execution of a machine level routine in memory. Where CALL simply executes the indicated routine, USR allows the BASIC program some control over the machine level routine by handing up to two bytes of information to the subroutine before commencing execution. Further, when the routine is finished, it passes a value back to the BASIC program.

The USR statement must be used within a larger BASIC statement, for example, X = USR(8) or PRINT USR(247). The value given within the parentheses is the decimal value of data passed to the subroutine, and after the subroutine is executed, USR( ) represents the value which is given back to the BASIC routine. Suppose that the two examples cited above use a subroutine which doubles any value it is given. Then the statement X = USR(8) would assign X the value 16, and PRINT USR(247) would print 494.

Listing 4–4 is an alteration of Listing 4–3. Key in Listing 4–4 and run the program. You will see the message

```
WHAT CHARACTER DO YOU WISH TO PRINT?
```

appear on the screen. Press any character key and then the RETURN key. After a short wait, the screen will fill with the character that you have chosen. When the screen has filled, a number is printed in the upper left corner of the screen; the "ALL DONE" message is printed near the bottom. Next, the message

```
ANOTHER CHARACTER? (Y/N)
```

will appear, giving you a chance to try another character.

```
10 REM DEMONSTRATE USR
20 REM
30 REM POKE THE MACHINE
40 REM LEVEL ROUTINE
50 REM INTO MEMORY
60 REM
65 POKE 10,76: POKE 11,0: POKE 1
   2,3
70 FOR L = 768 TO 805
80 READ V
90 POKE L,V
100 NEXT L
110 REM
120 REM POKES COMPLETE
130 REM
134 PRINT "WHAT CHARACTER DO YOU
    WISH TO PRINT?"
136 GET C$:C = ASC (C$) + 128
140 PRINT "GET READY"
150 FOR I = 1 TO 1000: NEXT I
160 PRINT USR(C)
170 VTAB (22): PRINT "ALL DONE"
172 PRINT "ANOTHER CHARACTER? (Y
    /N)": GET R$
174 IF R$ = "Y" GOTO 134
176 IF R$ = "N" GOTO 250
178 PRINT CHR$ (7): GOTO 172
180 REM
190 REM DATA TO POKE
200 REM
205 DATA 32,12,225
210 DATA 169,0,133,254,169,7,13
    3,255,160,255
220 DATA 162,4,32,88,252,165,16
    1,145,254
230 DATA 198, 254,208,252,136,2
    08,247,202,240,5,198,255,76
240 DATA 20,3,96
250 END
```

Listing 4-4.

The character which you select is passed to the subroutine via the USR statement. One caution: Pressing the "P" key while holding down the CTRL and SHIFT keys will generate a syntax error due to the constraints of the ASC function used in line 136. The number that appears at the top of the screen is caused by the statement "PRINT USR (C)", which actu-

ally prints the value returned by the subroutine. The subroutine in this example returns only garbage.

The following alterations are those which have been made to Listing 4-3:

Lines changed: 10, 70, 160, 170, 240
Lines added: 65, 134, 136,172,174,
176,178,205

These differences are explained below.

```
65 POKE 10,76: POKE 11,0: POKE 12,3
```

Line 65 places information into memory which is needed by the USR command. When BASIC encounters USR, it automatically jumps to location 10, and from there to the location of the subroutine which is to be used. The particular values POKEd here direct the computer to jump to location $300, where lies the subroutine. Location 11 contains the second byte of the subroutine's address, in this case $00, and location 12 contains the first byte—$03. If the subroutine begins in a different location, say $C025, the values 37 ($25 = 2*16 + 5) and 192 ($C0 = 12*16 + 0) would be POKEd into locations 11 and 12, respectively.

```
70 FOR L = 768 TO 805
```

The loop goes from 768 to 805 instead of to 802 because of the three extra data values which have been added in line 205. The effect of those three extra values is to accept the character code which is passed by the USR command.

```
134 PRINT "WHAT CHARACTER DO YOU
    WISH TO PRINT?"
136 GET C$:C = ASC (C$) + 128
```

These lines GET the character that the user wishes to print and convert it to its Apple code value.

```
160 PRINT USR(C)
```

This line passes the selected character code to the subroutine and then executes it. When the subroutine is finished, this statement also prints the value which is returned by the subroutine. In this case the value printed is meaningless and it will not be printed if line 160 is changed to read 160 X=USR(C).

```
170 VTAB (22): PRINT "ALL DONE"
```

The VTAB has been changed to 22 to allow program line 172 to print on screen line 23, and the GOTO has been removed since the program no longer ends at this point.

```
172 PRINT "ANOTHER CHARACTER? (Y
    /N)": GET R$
```

```
174 IF R$ = "Y" GOTO 134
176 IF R$ = "N" GOTO 250
178 PRINT CHR$ (7): GOTO 172
```

These lines inquire whether the user wants to try another character; if yes then GOTO line 134 to restart, if no then GOTO the END statement, and if neither then beep the bell and try for another response.

```
205 DATA 32,12,225
```

The three new data values discussed with line 70.

```
240 DATA 20,3,96
```

The first piece of data in this line is different than in Listing 4–3.

The subroutine which prints the characters for Listings 4–3 and 4–4 is only slightly faster than the BASIC program in Listing 4–1, which also fills the screen with a selected character. However, it is worth noting that the machine level subroutine used has been slowed down by a factor of approximately 125 so that the process will take a noticeable amount of time. To help demonstrate the speed of execution for machine level routines, replace each of the first four data values in line 230 with the value 234, to disable the delay function, and then run the modified program.

Vocabulary

CALL

PEEK

POKE

USR

Exercises

1. Write a BASIC program to determine the values at memory locations 1020, 1021, 1022, and 1023.
2. Write a BASIC program to poke the following values into memory locations 4620 through 4677.

   32, 88, 252, 169, 217, 32, 240, 253, 169, 207, 32, 240, 253, 169, 213, 32, 240, 253, 169, 160, 32, 240, 253, 169, 196, 32, 240, 253, 169, 201, 32, 240, 253, 169, 196, 32, 240, 253, 169, 160, 32, 240, 253, 169, 201, 32, 240, 253, 169, 212, 32, 240, 253, 169, 161, 32, 240, 253
3. Alter the program written in Exercise 2 so that it places the machine code values into memory and then executes them as a subroutine.

4. Write a BASIC program which plots several blocks or lines on the Low-Res screen, and then clears that screen by CALLing the ROM routine located at $F832.

5. Write a BASIC program which prints, on line 5 of the text screen, a message of your choice in the inverse mode, and then prints, on screen line 10, a second message in the normal mode. Do not use the VTAB, INVERSE, or NORMAL commands, but instead use the POKE and CALL commands. The VTAB location is discussed in the POKE section of this chapter, and the ROM routines to set the INVERSE and NORMAL modes are located at $FE80 and $FE84, respectively.

5

Graphics Modes and Soft Switches

Objectives

After reading Chapter 5 you should be able to:

- Set any of the graphics modes and their variations by using the soft switches.

Your Apple computer contains a chunk of hardware which is responsible for looking at portions of memory and translating what it finds there into a video display. Within limits, you may determine which area of memory is selected, and how the data there is interpreted. Data in memory may be interpreted for video display in three different ways: as Text, Low-Res (Low Resolution) graphics, or Hi-Res (High Resolution) graphics. Those three modes, their ten variations, and the method for selecting each will be discussed in this chapter.

Text Mode

The Text screen of an Apple II or Apple II+ comes from the factory able to display 24 lines of text on the screen with 40 characters across each line. It can display only 64 different characters; those are the 26 letters of the alphabet (upper case only), 28 special characters (such as parentheses, commas, plus signs, and so forth), and the ten numerals (zero through nine). There are several auxiliary hardware boards on the market today which connect through one of the interface slots in the back of the computer and allow for the display of lower case letters, more than 24 lines per

screen, and more than 40 characters per line. We will confine our discussions to the unmodified machine.

There are actually two Text screens which may be displayed—Text page 1 and Text page 2. From BASIC the TEXT command automatically displays the primary screen (page 1); there is no BASIC command to display the secondary screen (page 2). One reason for this is that the memory range displayed for page 2 is the same memory which is used to store APPLESOFT programs, and so all the BASIC programmer sees on page 2 is a jumbled picture of his or her program. Therefore, page 2 of text is of interest only if you are programming in assembly or machine code.

While in the Text mode, each character displayed is represented in memory by a numeric code. For example, $C1 (193) is the code for "A," $C2 (194) is the code for "B," and so on. The standard scheme used for assigning numeric codes to the characters is ASCII, an acronym for American Standard Code for Information Interchange. Unfortunately, your Apple does not use ASCII for its internal codes. The Apple value for any character displayed normally (as opposed to displayed as flashing or inverse) turns out to be $80 (128) greater than the ASCII value for that same character. Appendix 2 contains charts of both the ASCII and Apple codes for representing characters.

Low-Res Mode

As with Text, there are two different Low-Res screens, Low-Res page 1 and Low-Res page 2. Each page may be displayed as full graphics or as a mixed mode (graphics with four lines of text at the bottom of the screen). Like Text page 2, Low-Res page 2 uses the memory range where APPLESOFT stores programs, so the secondary page is of little use. The mixed screen Low-Res mode allows for color blocks to be displayed in 40 horizontal rows and 40 vertical columns (1600 blocks). Each block may be one of 16 colors, and four lines of text are displayed at the bottom of the screen. The full screen Low-Res mode replaces the four lines of text with an additional eight rows of blocks, so you have a total of 1920 blocks (40*48). The GR command sets the mixed screen, Low-Res graphics mode, page 1; there is no BASIC command to set the full screen, Low-Res mode, page 2.

Hi-Res Mode

The Hi-Res graphics mode also has a page 1 and a page 2 screen, both of which can be set to full graphics or mixed mode (graphics with four lines of text). There are 288 vertical columns of dots in 192 horizontal rows (or 53,760 total dots), on each full page of the Hi-Res graphics mode. HGR sets mixed mode, Hi-Res graphics, page 1; HGR2 sets full page, Hi-Res graphics, page 2. You cannot set any of the other two Hi-Res graphics modes directly from BASIC.

For those of you whose computers store APPLESOFT in RAM, Hi-Res graphics, page 2 is not available for use. It may, however, be displayed if you would like to see a "picture" of BASIC.

Soft Switches and You

The BASIC commands used to set the different display modes are suitable for many applications, but they do have their drawbacks. Besides the normal problem BASIC statements have of being slow to execute, BASIC has no direct way for you to see full page graphics on page one, mixed graphics on page two, or to switch back and forth between the text and graphics screens without clobbering the graphics. The solution lies in setting the proper graphics mode yourself instead of through BASIC, and that is where soft switches play their part.

The four switches that control the output mode are called soft because they are not physical switches, but rather they are features of the operating system. It is convenient to think of each soft switch as being a two-position toggle switch, as shown in Figure 5-1.

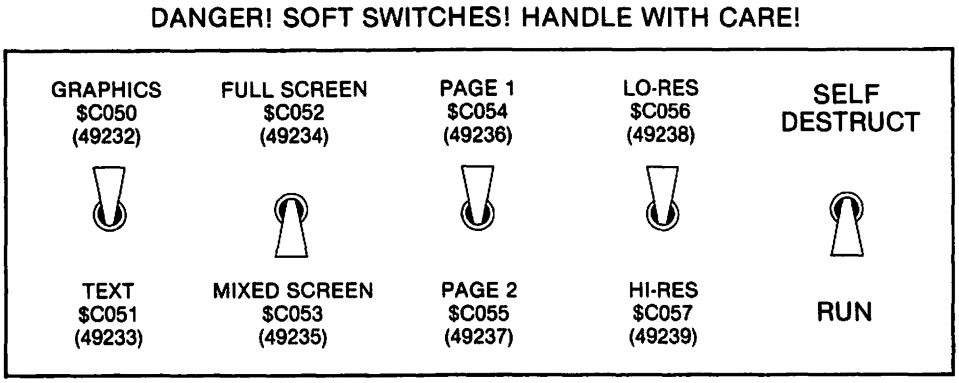


Figure 5-1. Toggle switches.

Each switch may be toggled in one of two different directions by referencing the associated memory location.

- Switch 1 determines whether the screen will display in the Text mode or one of the graphics modes.
- Switch 2 chooses between the full screen graphics mode or mixed screen graphics with four lines of text at the bottom. If switch 1 is set to Text, then switch 2 has no visible effect.
- Switch 3 selects which page will be displayed. Each display mode actually has two entirely different screens which may be used.

- Switch 4 selects between the Low-Res and Hi-Res modes. Switch 4 has no visible effect unless switch 1 is set to display graphics.

The APPLE IIe has several additional display-oriented soft switches. Those switches are discussed later in this chapter under the heading "For IIe's Only."

Toggling Soft Switches

Dismembering your computer to look for these switches will not help you locate them; remember, they are not a physical switch, but only exist in the system software. So how do you flip a switch that does not exist? When the computer is turned on, the operating system automatically toggles switch one and switch three to display the primary page of text, but from there it is up to you.

Each switch may be flipped in the desired direction by referring to the associated address. To set switch 1 from BASIC to display graphics, you could PEEK at the value which is in location 49232, or POKE a new value in there. Neither the value POKEd nor the the value PEEKed is of any consequence; POKE 49232,Ø, X = PEEK (49232), PRINT PEEK (49232), and POKE 49232,175 all set switch 1 to graphics mode. It is the mere act of referring to that location which sets the switch. From Monitor you can set the graphics switch by typing CØ5Ø <RETURN>, so that Monitor will display the value in location CØ5Ø, or type CØ5Ø:Ø to place a zero in that location. Again, it is only the reference to the location which is important.

Try setting the switches to view the first page of Hi-Res graphics without the four lines of text at the bottom and without clearing the screen. Refer to Figure 5–2 for the associated addresses for each switch. Your sequence of commands should look something like this. From the BASIC prompt (> or])type:

```
POKE 49234,Ø  (Set full screen)
POKE 49236,Ø  (Set page 1)
POKE 49239,Ø  (Set Hi-Res mode)
POKE 49232,Ø  (Set graphics)
```

The order is not important, but if you set the graphics mode first (POKE 49232,Ø), then the remainder of the POKEs will not be visible to you since the text is no longer being displayed.

To set the same display sequence from Monitor, type:

```
CØ52: Ø
CØ54: Ø
CØ57: Ø
CØ5Ø: Ø
```

Switching to text page 1 may be accomplished by using only two of the soft switches. You would type:

From BASIC	From Monitor
POKE 49236,Ø	CØ56: Ø
POKE 49233,Ø	CØ51: Ø

The first line sets switch 3 to display the primary page (page 1), and the second sets switch 1 to display text. The settings of switches two and four do not matter since they affect only the graphics modes.

A summary of the available video display modes appears in Figure 5-3. Practice setting each mode and its variations by the use of the soft switches; when you feel comfortable, proceed to the next chapter.

Switch 1	$CØ5Ø (49232)	Graphics
	$CØ51 (49233)	Text
Switch 2	$CØ52 (49234)	Full Screen
	$CØ53 (49235)	Split Screen
Switch 3	$CØ54 (49236)	Page 1
	$CØ55 (49237)	Page 2
Switch 4	$CØ56 (49238)	Low-Res
	$CØ57 (48239)	Hi-Res

Figure 5-2. A summary of soft switch addresses.

Text (always full page)
- *Page 1*
- *Page 2*

Low-Res
- *Page 1*
  - Mixed
  - Full page
- *Page 2*
  - Mixed
  - Full page

Hi-Res
- *Page 1*
  - Mixed
  - Full page
- *Page 2*
  - Mixed
  - Full page

Figure 5-3. Summary of available modes.

For IIe's Only

The APPLE IIe uses three additional soft switches to help control the display. These switches are referred to by the names **80COL, 80STORE,** and **ALTCHARSET.**

80COL determines whether the text is displayed with the normal 40 columns per line or with 80 columns per line. (You must have an 80 column card installed to display 80 columns per line!) To set the 40 column mode, POKE a value (any value) into 49164 ($C00C). To enable the 80 column mode, POKE location 49165 ($C00D).

80STORE determines whether addresses referencing memory locations within the first page of text (1024 to 2047) actually refer to memory on the main memory board, or memory on the auxiliary 80 column board. Do you have the feeling that you missed something? Let's back up a bit. Allowing an 80 column display doubles the number of characters which are displayed on one page of text. This requires twice the amount of memory. The Apple IIe designers had to find an additional 1024 bytes of memory somewhere, without disturbing the existing memory ranges used in the APPLE II+. The solution was to "clone" Page 1 of Text memory and put the duplicate memory on the 80 column board. But now there are two post office boxes with the address 1024, two boxes addressed 1025, and so on through 2047. When your computer "postman" is told to deliver a value to one of these addresses (for example, address 1777), he or she must know which of the two boxes having that address is to receive the value. The 80STORE switch acts as a mini ZIP code; it specifies which of the two banks of memory, main or auxiliary, is intended. POKEing any value into 49152 ($C000) will route all addresses to memory on the main board, and POKEing into 49153 ($C001) routes addresses from 1024 through 2047 to the memory on the 80 column board.

ALTCHARSET switches the character generator (the thing that takes the numeric code for an A and makes it appear as A on the on the screen) between the standard character set (American characters) and some on-board, alternate character set. For, though it may come as a surprise to

80COL	$C00C (49164)	40 Columns
	$C00D (49165)	80 Columns
80STORE	$C000 (49152)	Store in main memory
	$C001 (49153)	Store in auxiliary memory
ALTCHARSET	$C00E (49166)	Standard character set
	$C00F (49167)	Alternate character set

Figure 5-4. Summary of Apple IIe soft switches.

some Americans, not every country's alphabet consists of our 26 ABCs. This switch, however, does little for the BASIC graphics programmer, so we will not be discussing it further.

In addition to the added switches, the Apple IIe allows the programmer to determine the setting of any of the seven switches by PEEKing the contents of a related location. This way you can discover if the machine is set to Page 1 or Page 2, Text or Graphics, Hi-Res or Low-Res, and so forth, without actually POKEing any of the associated soft switches. For more discussion of these test locations, please refer to your APPLE IIe manual.

Vocabulary

ASCII

Full screen graphics

Hi-Res mode

Low-Res mode

Mixed screen graphics

Primary page

Secondary page

Soft switch

Exercises

Set the given display mode once using BASIC, and again using Monitor.

1. Text mode, page 2.
2. Full screen Low-Res mode, page 1.
3. Mixed screen Hi-Res mode, page 2.
4. Full screen Hi-Res mode, page 1.

6

Text and Low-Res

Objectives

After reading Chapter 6 you should be able to:

- Convert any pair of Low-Res colors into the corresponding value, both decimal and hex.
- Place any letter or pair of blocks at any given location on the Text or Low-Res screens, and do so using both Monitor and the POKE statement.
- Use the Low-Res editor to develop figures.
- Save the Low-Res screen using BSAVE or the Monitor WRITE command.

Although many programmers have an aversion to graphics in the Low-Res mode, a number of respectable programs have employed Low-Res with success. The "Little Brick Out" (on your DOS 3.3 system master) and "Lemonade" games are good examples of what can be done with Low-Res and a little imagination. We will demonstrate how the Text and Low-Res modes work, and how you may easily create and display Text and Low-Res figures.

Let's begin our exploration of Low-Res graphics by examining the Text mode a bit closer.

Text

To start with, type in Listing 6-1, which is designed to fill the screen with

the letter "A" without using a PRINT statement. It is similar in effect to Listing 4–3, but uses no machine language subroutine.

```
10 HOME
20 FOR I = 1024 TO 2047
30 POKE I,193: REM 193 IS "A"
40 FOR J = 1 TO 30 :REM DELAY
50 NEXT J
60 NEXT I
70 CALL 65338: REM BEEP SPEAKER
80 GOTO 80
```

Listing 6–1.

After clearing the screen, the program proceeds to POKE the numeric code for "A" into every memory location from 1024 through 2047 ($400–$7FF), which is the range displayed on the Text/Low-Res screen. You will see the screen fill with "A's"; the speaker will beep when the screen is full. In order to regain control of your machine you will have to press <RESET>.

Try replacing the number 30 in line 40 with a different value. The higher the number you select, the slower the screen will fill. Also try changing the value POKEd in line 30; for example, using a 1 instead of 193 will fill the screen with inverse "A's".

Notice that the screen fills in three separate pieces, even though the program fills memory sequentially. The first line of A's appears at the top of the screen, the second appears almost a third of the way down the screen, and the third line appears roughly a third below the second. The fourth line fills under the first, the fifth under the second, and so on until the screen is full. This three-part filling sequence is not just coincidence, but is more of a curse which haunts anyone dealing with graphics. It makes determining the address of any location on the screen rather difficult. The reason for the peculiar design of screen memory is tied up with the electronics, though in simple terms it was easier to wire the computer together that way. As you can see, however, locating the proper address to match a desired position on the screen could turn into a problem.

This problem is usually solved by the use of a memory map such as the one in Figure 6–1.

Finding Your Way

To use the map, first locate the position that you want on the map, say the second row down and the third box from the right. The address of that space on the text screen is found by adding the row and column numbers which are found to the left of the chosen box and directly above it, in this

example 1152 + 37 = 1189. The hex address is $480 + $25 = $4A5. (Remember: $8 + $2 = $A.)

Example 1

Use POKE to write "ACROSS" horizontally on the text screen, beginning in the third row and twenty-first column.

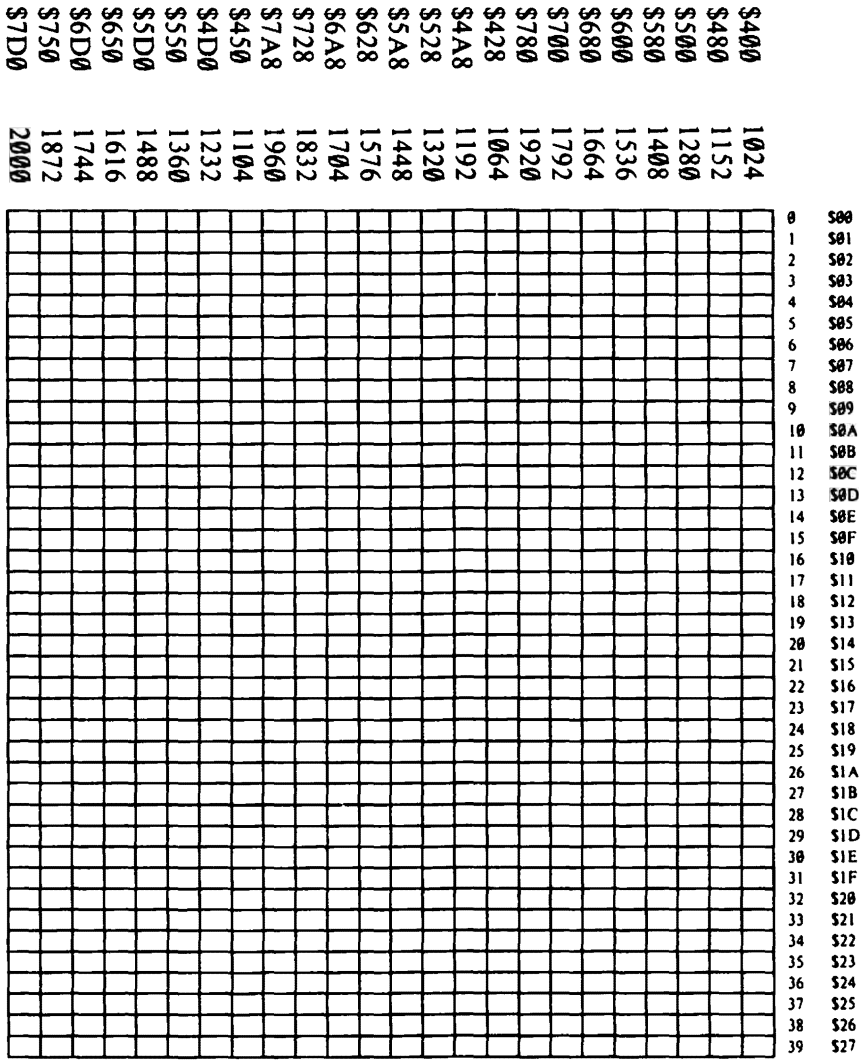


Figure 6–1. Text/Low-Res memory map.

(Used with permission of Apple Computer Corporation)

Solution

Row three, column twenty-one corresponds to the decimal address 13ØØ, so we will POKE the numeric values for the letters into consecutive memory locations beginning there. We must be careful that the screen does not scroll up while we are POKEing the values in, for if it does, the letters we place in memory will scroll also. To prevent the scrolling, we will begin by moving the cursor to the top of the screen. From BASIC enter:

```
CALL -936              (The HOME routine)
POKE 13ØØ,193          (A)
POKE 13Ø1,195          (C)
POKE 13Ø2,21Ø          (R)
POKE 13Ø3,2Ø7          (O)
POKE 13Ø4,211          (S)
POKE 13Ø5,211          (S)
```

*** A special note for you Apple IIe users: to make this example work as promised, you must have the computer in the 4Ø-column mode. To insure that it is, you may add the following two POKEs to the beginning of the preceding list.

```
POKE 49164,Ø           (Set 4Ø-column display)
POKE 49152,Ø           (Store values in main
                        memory)
```

Example 2

Use Monitor to place the message DOWN vertically on the screen beginning at row two, column twenty-four, and in FLASH mode.

Solution

The vertical addresses are not consecutive, but with the aid of the memory map and the Apple character chart the proper values are determined. Type:

```
CALL -151
25: Ø                  (move cursor to top)
497: 44                (D)
517: 4F                (O)
597: 57                (W)
617: 4E                (N)
```

Lost and Found

While running the program (Listing 6–1), you may have also noted that after each line drawn on the bottom third of the screen, there was a slight pause before the filling continued on the upper third. That happens because there are eight bytes of "phantom storage" following each line in the lower portion. Those bytes exist in memory, but are not displayed on the screen. To illustrate, calculate the address of the right-most box in the second row from the bottom of Figure 6–1. You should find it to be 1872 + 39, or 1,911. The memory that follows that is the eighth line from the top which commences with location 1,92Ø, so locations 1,912 through 1,919 are "lost." Lost, but not forgotten, for Apple has put that wasted memory to use within DOS to remember which drive was most recently accessed. As a matter of fact, the next time that you use your disk after running this program, you will notice that your disk makes that fearsome grunting noise as it recalibrates.

There are other games to play with Text, such as saving the complete screen, but since Text is so similar to Low-Res, we might as well discuss those tricks under the more colorful guise of ....

Low-Res Graphics

Although Low-Res graphics are not as popular as they once were, they still have their uses. Low-Res may be the proper option if your major concerns are simplicity, speed, color, and memory conservation; and if the blocky nature of the mode is not a debilitating obstacle to you.

Low-Res graphics use the same area of memory and the same memory map as Text, so use of the two modes is similar. Alter the program in Listing 6–1 by changing line 1Ø to read:

```
1Ø GR: REM SET GRAPHICS DISPLAY
```

Run the modified program. The graphics screen will be filled with pairs of little boxes—a magenta box on top of a green box. (The precise colors may vary with individual TV sets.) The text window at the bottom of the screen will still fill with "A's."

Add a line to the program:

```
15 X = PEEK (49234)
```

and run it again. Line 15 sets the soft switch to display full screen graphics, so you will see the pairs of blocks all the way down the screen with no text window.

How'd You Do That?

Line 3Ø POKEs the value 193 ($C1) into every address in Low-Res memory.

Each byte of Low-Res memory determines two colored blocks. Refer to the Low-Res color chart in Figure 6–2, and you will find that $C is the Low-Res code for the light green block, while $1 is the code for the magenta block. In Low-Res, the bottom block is determined by the left-hand nibble, and the top block is determined by the right-hand nibble. Using only a nibble to determine each block leads to some inconvenience because memory can be controlled one byte at a time, and no less. This means that calculating the correct combination of two nibbles for the proper two boxes is often confusing.

$Ø (Ø) BLACK	$8 (8) BROWN
$1 (1) MAGENTA	$9 (9) ORANGE
$2 (2) DARK BLUE	$A (1Ø) GREY
$3 (3) PURPLE	$B (11) PINK
$4 (4) DARK GREEN	$C (12) GREEN
$5 (5) GREY	$D (13) YELLOW
$6 (6) MEDIUM BLUE	$E (14) AQUA
$7 (7) LIGHT BLUE	$F (15) WHITE

Figure 6–2. Low-Res color codes.

Example 3

Suppose that you want to place a grey block over a yellow block at location 1Ø28. First, you must calculate the value that will give you the desired combination. Again referring to Figure 6–2, yellow is color number $D and grey is number $5. Arrange the digits as $D5 so that grey will be on top, and convert $D5 to decimal 213. From BASIC type:

```
GR
POKE 1Ø28,213
```

The grey and yellow pair of blocks will appear near the upper left corner of the screen. You can use Monitor to achieve the same end by using the hex equivalent of the numbers. Still from BASIC, type:

```
GR
CALL -151
4Ø4: D5
```

Experiment with Low-Res by plotting various combinations of colored blocks in different locations.

Creating Low-Res Pictures

Creating a drawing in either Low-Res or Hi-Res requires a good deal of

preliminary planning. We will take you through the process of creating a Low-Res apple (what else?!).

The first step is to make a sketch of the figure, in this case an apple. Those of you that claim no artistic ability can find patterns to trace—children's coloring books and cross-stitch or embroidery patterns make good sources of simple figures. Figure 6–3 represents the preliminary sketch of the apple.

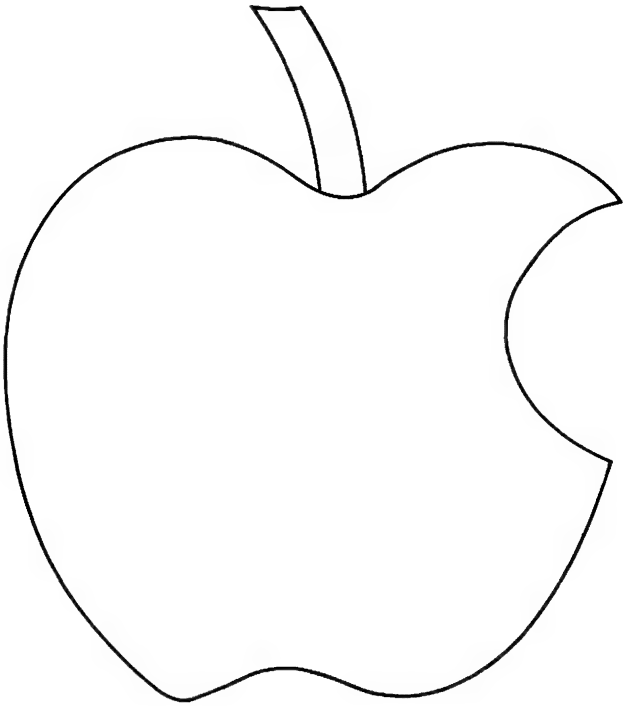


Figure 6–3. An apple.

After your sketch is completed to your satisfaction, the outline must be blocked to fit with the Low-Res bricks as in Figure 6–4. Notice that the leaf was omitted in the blocked figure; it proved impossible to block in a recognizable manner. It is better to identify that type of problem before you begin working with the computer!

Blocking is not as simple as it might be since the Low-Res blocks are not square. The height of each brick is approximately two-thirds its length, and that distortion must be accounted for. Perhaps the simplest technique for proportioning the blocks is to use very fine graph paper—perhaps ten squares to the inch—and define each block to be three squares long and two squares high. Place your sketch over the graph paper and go over the outline again while applying firm pressure with the pen. This will leave an impression of your outline on the blocking paper, and then you may square up the outline as in Figure 6–4.

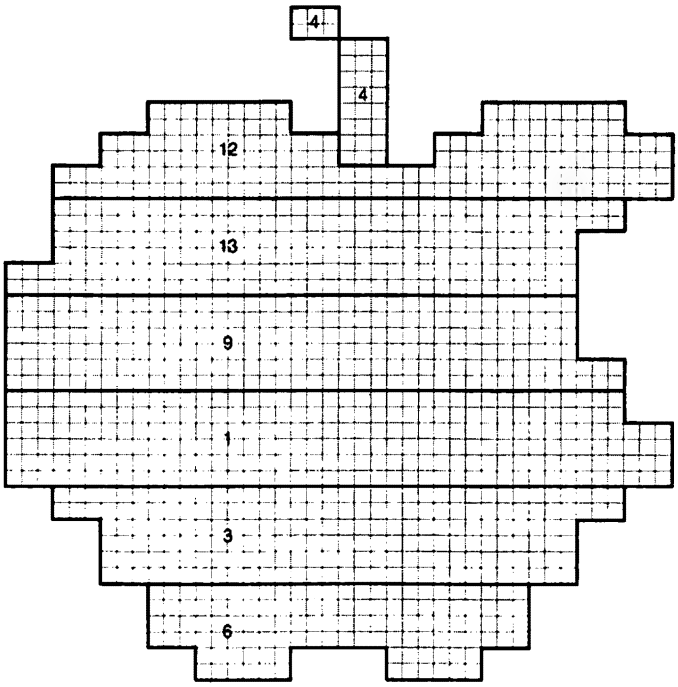


Figure 6-4. Apple block diagram.

Next, decide which of the Low-Res colors you wish to use in your apple and indicate them on your block drawing. With all of that done, you are now ready to begin putting the picture onto the screen. There are two methods of accomplishing this. The first technique is to make use of the PLOT, HLIN, and VLIN statements and place those within your program; the second involves using a Low-Res editor program (described later in this chapter) to plot the apple directly onto the screen, and then record the contents of Low-Res memory.

Hatching a Plot

The brute-force method uses the PLOT, HLIN, and VLIN commands; simply sit down with the blocked figure and develop a list similar to Listing 6-2 which creates the apple by drawing an obnoxious number of horizontal lines.

```
1Ø REM PLOT APPLE
```

```
20 REM
30 GR
40 COLOR= 4
50 PLOT 20,10
60 VLIN 11,14 AT 21
70 COLOR= 12
80 HLIN 17,19 AT 13
90 HLIN 24,26 AT 13
100 HLIN 16,20 AT 14
110 HLIN 23,27 AT 14
120 HLIN 15,27 AT 15
130 COLOR= 13
140 HLIN 15,26 AT 16
150 HLIN 15,25 AT 17
160 HLIN 14,25 AT 18
170 COLOR= 9
180 HLIN 14,25 AT 19
190 HLIN 14,25 AT 20
200 HLIN 14,26 AT 21
210 COLOR= 1
220 HLIN 14,26 AT 22
230 HLIN 14,27 AT 23
240 HLIN 14,27 AT 24
250 COLOR= 3
260 HLIN 15,26 AT 25
270 HLIN 16,25 AT 26
280 HLIN 16,25 AT 27
290 COLOR= 6
300 HLIN 17,24 AT 28
310 HLIN 17,24 AT 29
320 HLIN 18,19 AT 30
330 HLIN 22,23 AT 30
```

Listing 6-2. Apple drawn using VLIN and HLIN.

The PLOT was used for the top, VLIN for the remainder of the stem, and since the colors in the apple run horizontally, HLIN was used for the rest of the figure. Which of the commands you use most will depend solely on the figure that you want to draw. This technique has obvious drawbacks if the picture is composed of many isolated blocks and few horizontal or vertical lines. In any event, it is quite tedious for the programmer; an aspect only partially balanced by the efficiency with which the program is executed.

A variation on this technique is to place an HLIN statement within a loop and read the individual numbers from a data table as shown in Listing 6-3.

```
10 GR
20 COLOR = 4
```

```
30 PLOT 20,10
40 VLIN 11,14 AT 21
50 COLOR = 12
60 FOR I = 1 TO 5
70 READ X1,X2,Y
80 HLIN X1,X2 AT Y
90 NEXT I
100 COLOR = 13
110 FOR I=1 TO 3
        .
        .
        .
350 DATA 17,19,13,24,26,13,16,
    20,14,23,27,14,15,27,15
360 DATA 15,26,16,15,25,17,...
        .
        .
        .
```

Listing 6-3.

Although this is easier for the programmer, the advantage of speedy execution has been sacrificed. Unless the program is to be run only a few times, it is usually better for the programmer to expend the extra effort in order to enhance the efficiency of the execution.

Dealing with Memory

The basic scheme behind this second technique and its variations is to draw your figure on the screen initially, and then save the contents of Low-Res screen memory. Then, when you wish to use that figure in a program, you need only replace the contents of memory with the data already saved.

The first step is to draw your picture on the screen. This may be done using PLOT, HLIN, and VLIN as in the previous discussion, or by using an editor. An editor is a program which simplifies creating, altering, and saving either text or graphics. A sample Low-Res editor is provided in Listing 6-4. The editor will let you plot and erase blocks in any of the Low-Res colors anywhere on the screen, except in the bottom four lines of text.

```
10 HOME : GR
20 PF = 0:X = 20:Y = 20
30 GOSUB 230
40 C = PEEK (49152): IF C > 127 THEN
   C$ = CHR$ (C-128): POKE -16368,0:
   GOTO 60
50  PLOT X,Y: COLOR= 0: PLOT X,Y:
   COLOR= CN: GOTO 40
```

```
60 REM
70 IF PF THEN PLOT X,Y:PF = 0
80 IF C$ = "J" AND X > 0 THEN X = X-1:
   GOTO 190
90 IF C$ = "I" AND Y > 0 THEN Y = Y-1:
   GOTO 190
100 IF C$ = "K" AND X < 39 THEN X = X + 1:
    GOTO 190
110 IF C$ = "M" AND Y < 39 THEN Y = Y + 1:
    GOTO 190
120 IF C$ = "U" AND X > 0 AND Y > 0 THEN
    X = X-1:Y = Y-1: GOTO 190
130 IF C$ = "O" AND X < 39 AND Y > 0 THEN
    X = X + 1:Y = Y-1: GOTO 190
140 IF C$ = "," AND X < 39 AND Y < 39 THEN
    X = X + 1:Y = Y + 1: GOTO 190
150 IF C$ = "N" AND X > 0 AND Y < 39 THEN
    X = X-1:Y = Y + 1: GOTO 190
160 IF C$ = "P" THEN PF = 1: GOTO 190
170 IF C$ = "Q" THEN GOTO 280
180 IF C$ = "C" THEN GOSUB 230
190 GOTO 40: REM BACK TO TOP OF LOOP
200 REM
210 REM SET COLOR
220 REM
230 VTAB (24): INPUT "NUMBER FOR NEW
    COLOR:?";CN$
240 CN = VAL (CN$): IF CN < 0 OR CN > 15
    THEN PRINT CHR$ (7): GOTO 230
250 PRINT : PRINT : PRINT : PRINT
    "COLOR = ";CN
260 COLOR= CN
270 RETURN
280 REM
290 REM
300 PRINT "QUIT? (Y/N)";: GET R$
310 IF R$ <> "Y" GOTO 40
320 END
```

Listing 6-4. Low-Res editor.

This editor is not elaborate, but it does give you the functions necessary to plot and edit pictures on the Low-Res screen, and it is easily modified to suit your individual requirements.

When you run the editor, you will first be asked to enter the number of the color to be plotted. The computer will ask:

```
NUMBER FOR NEW COLOR:?
```

Type the decimal number for the color you want (see Figure 6-2) and then press return. You will then see a flashing cursor in the center of the screen. The cursor may be moved anywhere on the screen by using the keys shown below.

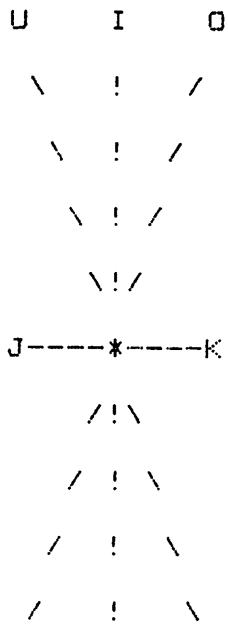


Figure 6-5. Cursor movement.

The I, M, J, and K keys move the cursor up, down, left, and right; the U, O, N, and , keys move the cursor diagonally as indicated. Try it!!

The editor also uses the P, C, and Q keys. To plot a point, press P and then move the cursor out of the way. Press C and you will be given the opportunity to set the color as you did at the beginning of the program. Press Q, and you will be asked

```
QUIT? (Y/N)
```

Press Y if you wish to quit and N if you do not.

Any color block may be erased by simply covering the offending block with the cursor. That makes erasing very simple—sometimes too simple, but you will get used to it.Use the editor to copy the apple from Figure 6-4.

Saving Figures

Now that you have a picture on the screen, you probably would like to be able to save it. Assuming that you have a disk drive, type

```
BSAVE APPLE,A$400,L$400
```

This statement causes the computer to save $400 bytes of memory beginning with location $400. The A parameter indicates the starting address, and the L parameter gives the length of the memory range to be BSAVE. The Low-Res screen memory begins at $400 and is $400 bytes long. If you do not care to work in hex, you may type

```
BSAVE APPLE,A1024,L1024
```

and do the same thing since $400 equals 1024.

With the screen safely saved on the disk, you may recall it at any time. To demonstrate, type:

```
BLOAD APPLE
```

and the apple will reappear on the screen. The A and L parameters may be used with BLOAD, but they are not necessary.

You may BLOAD the picture from within a BASIC program by inserting a line in the program similar to line 15 below. Load the Low-Res editor and type the following:

```
15 PRINT CHR$(4)"BLOAD APPLE"
```

When you run the editor it will now load the apple picture automatically. This is handy if you wish to re-edit an existing picture.

If you do not have a disk drive, you can save the screen memory on cassette tape by using the Monitor. Type:

```
CALL -151
400.7FFW
```

to tell Monitor to write (hence the W) the range of memory from $400 through $7FF onto the tape. To insure an accurate recording, it is a wise idea to put the tape on "record" and let it run for a few seconds between pressing the W key and the return key. After the Low-Res screen is saved on the tape, you can rewind the tape and load it from Monitor by typing

```
400.7FFR
```

The format is the same as the Write command, except the W is replaced by the R or READ command.

If you have saved and loaded the screen as suggested, you may have noticed that when the screen is loaded, it brings with it some garbage in the text window. The BSAVE command saves the whole screen, including whatever text was in the text window at the time. That may be circumvented by writing a short BASIC program to first clear the text window and then save the screen. That program is left to the reader as an end of the chapter exercise.

Vocabulary

BLOAD

BSAVE

Editor

HIMEM

HLIN

PLOT

VLIN

Exercises

1. Convert the following Low-Res color pairs into the hex value necessary to plot the first over the second.
   a. dark blue/green
   b. green/light blue
   c. aqua/pink
   d. orange/purple
   e. brown/black
   f. black/brown
2. Convert the color pairs in Exercise 1 to their decimal values.
3. Use the POKE statement to plot the following color pairs at the given location. The location is given by its row and column on the screen and memory map (see Figure 6–1).
   a. white/green at row 5, column 2Ø
   b. orange/black at row 19, column 1Ø
   c. aqua/pink at row 1Ø, column 12
4. Use Monitor to perform the plotting discussed in Exercise 3 above.
5. Use the POKE command to place the message "HI-SCORE" in the bottom right corner of the Low-Res text window.
6. Write a BASIC program to BSAVE the Low-Res screen after clearing the text window.
7. Design and plot a picture of a spaceship on the Low-Res screen.
8. Save the figure plotted in Exercise 7 using BSAVE (or the Monitor Write command).

7

Preserving Your Pictures

Objectives

After reading Chapter 7 you should be able to:

- Save the Low-Res screen using BSAVE, print a list of values, write the list of values to a text file on disk, or create a text file of DATA statements on disk.
- Replace a figure on the Low-Res screen by using BLOAD, read the text file of values from disk and POKE them into screen memory, POKE values from DATA statements—either keyed or EXECed into the program, or load the screen values into a safe memory area and then copy those values to Low-Res screen memory.

In Chapter 6 you discovered how to save the Low-Res screen using BSAVE or the Monitor WRITE command. Both methods were simple, but they both have the same drawback—they often save a great deal of empty space. Since disk space is a precious commodity, we will devote this chapter to discussing alternate methods for saving your pictures.

Scanning Memory

One alternate method of saving your picture is to record which bytes of Low-Res memory are turned "on," and what values they contain. In the apple picture in Chapter 6, only about one of every ten bytes contains a non-zero value. That is, nine-tenths of the screen is blank, and we are interested only in the non-blank portions—those that are displaying a

color. The following program searches through Low-Res memory and prints a list giving the location of every non-zero byte and its address. The program presumes that your printer is in slot 1, but that may be changed by altering the value assigned to SL in line 40. If you do not have a printer, set SL equal to 0.

```
10 REM PRINT LOW-RES VALUES
20 REM
30 REM
40 SL = 1: REM SL = PRINTER SLOT #
50 PR# SL
60 REM
70 REM
80 FOR I = 1 TO 20
90 READ S: REM STARTING ADDRESS OF
   SCREEN LINE
100 FOR L = S TO S + 39
110 IF PEEK (L) < > 0 THEN PRINT L;
    ", "; PEEK (L):COUNT = COUNT + 1
120 NEXT L
130 NEXT I
140 PRINT "COUNT= "; COUNT; "EOJ"
150 PR# 0
160 END
170 REM
180 REM DATA TABLE
190 REM
200 DATA 1024,1152,1280,1408,1536
210 DATA 1664,1792,1920,1064,1192
220 DATA 1320,1448,1576,1704,1832
230 DATA 1960,1104,1232,1360,1488
```

Listing 7-1. Low-Res scan #1.

The program uses a data table to straighten out the convolutions of screen memory. Each piece of data represents the starting address for a line on the screen (see Figure 6-1), and the 40 bytes beginning with each of those addresses are scanned by the FOR-NEXT loop in lines 100 through 120. This method scans the screen from top to bottom, and without scanning the memory which is not displayed. Line 110 prints the location (L) and contents (PEEK L) of any non-zero byte, and increments a counter. Only the top 20 lines are scanned, so none of the values in the text window are printed. After the scanning is complete, line 140 prints the number of "on" bytes, and an end of job message. Scanning the apple will give you a list of close to 109 addresses and values, depending on where you placed your apple on the screen.

Try the program out by loading it into memory and then typing:

```
GR
BLOAD APPLE
RUN
```

This will (1) set the graphics screen, (2) load the picture of the apple back onto the screen from disk, and (3) run the program in memory (hopefully Listing 7-1). The list of addresses and values will appear on your printer. If it does not, check your program against Listing 7-1 and make sure that you have the printer slot assigned correctly in line 40.

What to Do with Data

The list of addresses and values printed by the Low-Res scan #1 may be written into a program as a data table and POKEd back into memory using a BASIC subroutine similar to:

```
1000 FOR I = 1 TO 107
1010 READ L,V: POKE L,V
1020 NEXT I
1030 RETURN
1040 DATA 1684,4,1685,64,1809,192, ...
```

Listing 7-2.

The data values in line 1040 are those generated from Listing 7-1. This method has the advantage of not using up disk space, but it has probably occurred to you that entering all of those data values by hand is very inefficient, not to mention boring and prone to error. We have two alternatives for those of you with a disk drive.

The Data File

Instead of printing the addresses and values on paper, you can write them to a text file on the disk. Load Low-Res Scan #1 (Listing 7-1) and then type the following lines to modify it.

```
10 REM WRITE DATA FILE
35 D$ = CHR$ (13) + CHR$ (4)
40 INPUT "FILENAME:";FL$
45 PRINT D$ "OPEN" FL$
50 PRINT D$ "DELETE" FL$
55 PRINT D$ "OPEN" FL$
60 PRINT D$ "WRITE" FL$
110 IF PEEK (L) < > 0 THEN PRINT L:
    PRINT PEEK (L)
140 PRINT "*"
150 PRINT D$ "CLOSE" FL$
155 PRINT CHR$(7): REM BEEP SPEAKER
```

Listing 7-3. Low-Res Scan #2 (modifications only).

This modification will create a data file on the disk which contains the same list of addresses and values that you printed out with Low-Res Scan #1. Line 40 requires that you give the data file a name, and line 140 prints an "*" to mark the end of the file. To test the program, load it into memory and type:

```
GR
BLOAD APPLE
RUN
```

So that your filename is the same as the one used in the next example, enter APPLE.DTA when the computer requests a filename. After that, the disk will spin intermittently as the screen memory is being scanned and saved; the speaker will beep to signal the end of the run.

The data may be read from the file APPLE.DTA and POKEd into the proper locations by using a subroutine such as:

```
1000 GR
1010 D$ = CHR$ (13) + CHR$ (4)
1020 FL$ = "APPLE.DTA"
1030 PRINT D$ "OPEN" FL$
1040 PRINT D$ "READ" FL$
1050 INPUT L$
1060 IF L$ = "*" GOTO 1110: REM THE
     * MARKS THE END OF THE FILE
1070 L = VAL (L$)
1080 INPUT V
1090 POKE L,V
1100 GOTO 1050
1110 REM
1120 PRINT D$ "CLOSE" FL$
1130 RETURN
```

Listing 7-4.

To test this subroutine, enter it and these extra lines.

```
10 REM YOUR PROGRAM
20 GOSUB 1000
30 END
```

When you run the program, the graphics screen will be set and cleared, then the disk will spin, and the apple will appear. If your data file is not called APPLE.DTA, change line 1030 to assign FL$ the proper name.

This method is nice because you do not have to key in the large amounts of data, but unfortunately disk access is very slow and delays the execution of the program which uses the graphics. Moreover, this procedure not only requires that the user have a disk drive, but also opens a Pandora's box of problems such as having the disk in a different drive or

slot. To remedy those problems we present a slightly more complex technique.

Using EXEC

What you will do here is create a text file of DATA statements on disk which may be EXECed into any program that you like. Each record in the text file will contain a line number followed by the DATA statement which, in turn, is followed by ten pairs of entries which are the locations and values of the bytes used in screen memory. The last DATA statement may contain less than the 20 entries the others have. The line numbers for the DATA statements will begin at 10000 and increment by two; the set of values in the DATA statements will be the same as those printed with Low-Res Scan #1.

Load the program from Listing 7-1 once again, then enter the following statements to modify it.

```
10 REM CREATE EXEC FILE
35 D$ = CHR$ (13) + CHR$ (4)
40 INPUT "FILENAME:";FL$
45 PRINT D$ "OPEN" FL$
50 PRINT D$ "DELETE" FL$
55 PRINT D$ "OPEN" FL$
60 PRINT D$ "WRITE" FL$
70 LN% = 10000:C1 = 1
104 IF PEEK (L) = 0 GOTO 120
106 COUNT = COUNT + 1
108 IF C1 = 1 THEN PRINT LN%; "DATA";
110 IF C1 > 1 THEN PRINT ", ";
112 PRINT L;","; PEEK (L);:C1 = C1 + 1
114 IF C1 > 10 THEN C1 = 1:LN% = LN% + 2:
    PRINT
140 PRINT D$ "CLOSE" FL$
150 PRINT CHR$ (7); "COUNT =" ;COUNT; "EOJ"
```

Listing 7-5. Low-Res Scan #3 (modifications only).

Line 40 requires that you enter a name for the file being created. Line 70 sets the line numbers to begin at 10000 (LN% = 10000), and line 114 increments each line number by 2 (LN% = LN% + 2).

With this routine in memory, create the text file for the apple figure by typing:

```
GR
BLOAD APPLE
RUN
```

When asked for a filename, enter APPLE.EXC. The disk will behave very

much like it did when you created the data file using the previous method. When the program is finished, the speaker will beep and the computer will print the total number of bytes that it saved—write this number down someplace. APPLE.EXC is a text file which contains a number of DATA statements, complete with line numbers. Those DATA statements may be easily appended to your programs. To demonstrate, type:

```
NEW
10 GR
20 COUNT = xxx
30 FOR I = 1 TO COUNT
40 READ L,V: POKE L,V
50 NEXT I
100 END

EXEC APPLE.EXC
```

The value assigned to COUNT in line 20 should be the number you wrote down after running Listing 7-5, for that is the number of data bytes in the apple figure.

When the disk stops spinning, type:

```
LIST
```

and you will see the program lines you entered above plus several DATA statements beginning at line 10000.

When you are ready, type:

```
RUN
```

and the computer will draw the apple on the Low-Res screen.

Since the DATA lines are now part of the code, the program does not need to access the disk for the locations and values. This eliminates the difficulties inherent in using disk data files.

We have already stored the data for a figure in two places—the disk and the program code. There is yet another place to store that information, and that is within the computer itself.

Moving Memory

What we will do is store the entire apple screen in memory somewhere other than the Low-Res screen memory area. Then when you wish to display the figure, you need only copy the information from the storage area to Low-Res memory. We will perform the copying function using a BASIC routine, and then also do it using a machine language subroutine.

The first task is to load the apple figure into a safe, unused memory range. We will use the top $400 bytes of user memory, from $9200 to $95FF (37376—38399). The addresses given here are for a 48K Apple II+,

and will need to be adjusted if you have less memory in your machine. You may determine the highest usable memory address by referring to the Apple DOS 3.3 manual, page 142, or by booting your system and typing:

```
PRINT PEEK(115) + 256*PEEK(116)
```

If your computer is an Apple II with INTEGER BASIC in ROM, you would type:

```
PRINT PEEK(76) + 256*PEEK(77)
```

The number you get as a result is the address of the "top" of available memory. To fit your Low-Res screen data in below that location, subtract 1Ø24 ($4ØØ) to get the starting address of your storage area. Multiple screens may be stored one below the other by allowing 1Ø24 bytes for each screen.

Loading the apple figure at $92ØØ is easily accomplished by:

```
BLOAD APPLE,A$92ØØ
```

or

```
BLOAD APPLE,A37376
```

If you have the apple figure stored on tape, enter Monitor and type:

```
92ØØ.95FFR
```

After the apple has been loaded in the proper location, you need to reset HIMEM so that your BASIC program does not overwrite your data and reduce your picture to applesauce! From BASIC type:

```
HIMEM:37375
```

To get a preview of the speed of moving memory, enter Monitor and type:

```
4ØØ<92ØØ.95FFM
```

and, thanks to the Monitor MOVE command, you will see the apple ... in a flash!

This entire procedure was accomplished in your computer's immediate execution mode, but now we need a program to do the same things. Listing 7-6 is designed to do just that from APPLESOFT.

```
5 REM MOVE THE APPLE
6 REM USING POKE
7 REM
1Ø HIMEM: 37375
2Ø D$ = CHR$ (13) + CHR$ (4)
3Ø PRINT D$; "BLOAD APPLE,A37376"
4Ø SOURCE = 37376
5Ø GR
6Ø FOR I = Ø TO 1Ø23
```

```
7Ø POKE 1Ø24 + I, PEEK (SOURCE + I)
8Ø NEXT I
9Ø END
```

Listing 7-6. BASIC memory move.

Line 1Ø sets HIMEM down below the area needed to store our apple data in order to create a safe area of memory. Line 3Ø loads the data into that protected area, and line 4Ø sets the variable SOURCE equal to the address of the first byte of data. After the graphics mode is set, lines 6Ø through 8Ø read the data from the source area and POKE it into the screen memory area. When I equals Ø, line 7Ø PEEKs at the beginning address of the data (SOURCE + Ø equals 37376), and POKEs that value into the first byte of screen memory (1Ø24 + Ø). When I equals 1, line 7Ø PEEKs at the second byte of the data (SOURCE + 1 equals 37377) and POKEs that value into 1Ø25. The loop continues until I equals 1Ø23, at which time the contents of location 38399 is POKEd into location 2Ø47.

When you run the program, you will see that it does as promised, but the time needed to complete the memory transfer is gruesome.

Once again a machine language routine comes to the rescue. Run the program from Listing 7-7 (a modification of Listing 7-6). It POKEs a machine code subroutine into memory in line 4Ø, and the address of the data area in line 45. Line 7Ø CALLs the routine, and the apple appears right before your bloodshot little eyes! This routine is very convenient to use in your own programs because the subroutine and one or more pictures may be loaded in advance, and displayed quickly and easily by using the commands in lines 45 and 7Ø.

```
5 REM MOVE THE APPLE
6 REM USING MACHINE CODE
7 REM
1Ø HIMEM: 37375
2Ø D$ = CHR$ (13) + CHR$ (4)
3Ø PRINT D$; "BLOAD APPLE,A37376"
4Ø FOR L = 768 TO 795: READ V:
   POKE L,V: NEXT L
45 POKE 252,Ø: POKE 253,146
5Ø GR
6Ø REM GET READY
7Ø CALL 768
8Ø REM TA DA!!
9Ø END
1ØØ DATA 169, Ø, 133, 254, 168, 169,
    4, 133, 255, 17Ø, 177, 252, 145
11Ø DATA 254, 2ØØ, 2Ø8, 249, 2Ø2, 24Ø,
    7, 23Ø, 253, 23Ø, 255, 76, 1Ø, 3, 96
```

Listing 7-7. Memory move using CALL.

The POKEs in line 45 are very important because they provide the subroutine with the starting address of the data area. If the data area begins at a different location, then you will have to adjust these POKEs. The way to calculate the values is to write down the starting address in hex ($92Ø0), separate the two bytes so that it looks like 92 Ø0, and convert each byte to decimal: $92 = 9*16 + 2 = 146, $Ø0 = Ø. Those are the two values to be POKEd into 252 and 253, but they must be POKEd in the correct order or strange things will happen. Apple wants you to give it the right-hand (low order) byte first and then the left-hand (high order) byte—a conspiracy known as Lo-byte/Hi-byte—so the Ø is placed in 252 and the 146 in 253.

Example 1

Suppose that the subroutine in Listing 7-7 has already been POKEd into memory, and that you want to display a picture which is stored at $8EØC.

```
$8E = 8*16 + 14 = 142
$ØC = Ø*16 + 12 = 12
```

so you would

```
POKE 252,12: POKE 253,142
CALL 768
```

A further note of caution: The starting address must be POKEd just prior to CALLing the subroutine. APPLESOFT uses location 253 for scratch pad memory, so after the CALL is executed BASIC clobbers whatever value you had in there.

Sound Assembly, Bugler

If you speak assembly language, Listing 7-8 may be of interest. It is the assembly listing of the subroutine used in Listing 7-7.

```
                SOURCEL     EPZ     $FC
                SOURCEH     EPZ     $FD
                DESTL       EPZ     $FE
                DESTH       EPZ     $FF
                ;
                ;INITIALIZATION
                ;
Ø3ØØ- A9 ØØ                 LDA     #ØØ
Ø3Ø2- 85 FE                 STA     DESTL
Ø3Ø4- A8                    TAY
Ø3Ø5- A9 Ø4                 LDA     #Ø4
```

```
Ø3Ø7- 85 FF              STA     DESTH
Ø3Ø9- AA                 TAX
                ;
                ;BEGIN LDA/STA LOOP
                ;
Ø3ØA- B1 FC  LOOPTOP     LDA     (SOURCEL),Y
Ø3ØC- 91 FE              STA     (DESTL),Y
Ø3ØE- C8                 INY
Ø3ØF- DØ F9              BNE     LOOPTOP
Ø311- CA                 DEX
Ø312- FØ Ø7              BEQ     STOP
Ø314- E6 FD              INC     SOURCEH
Ø316- E6 FF              INC     DESTH
Ø318- 4C ØA
Ø3                       JMP     LOOPTOP
Ø31B- 6Ø     STOP        RTS
                         END
```

Listing 7-8. Assembly language subroutine.

In essence, this routine takes $4ØØ bytes of memory from the source range and copies it into the Low-Res page 1 area.

Wrapping Up

By now you have probably learned more than you ever wanted to know about Low-Res graphics, but do not despair, as many of the same concepts will apply in Hi-Res graphics as well. We have shown you how to control the individual units of the Text and Low-Res screens, and how to create, save, and recall pictures on the Graphics screen. Each of the various techniques discussed has its peculiar advantages and disadvantages—some are faster than others, some use more memory, some require disks, and so forth. It is up to you, the programmer, to choose the method best suited to your application.

Vocabulary

BLOAD
BSAVE
Editor
EXEC
HIMEM
Lo-byte/Hi-byte
PLOT

Exercises

1. Design and plot a picture of a spaceship on the Low-Res screen.
2. Save the figure plotted in Exercise 1 using:
    a. The method of Low-Res scan #2 (Listing 7-2).
    b. The method of Low-Res scan #3 (Listing 7-3).
3. Draw the figure saved in Exercise 2 by:
    a. Using BLOAD (or Monitor's READ command).
    b. Writing a BASIC program to input and plot the data generated in Exercise 2a.
    c. EXECing the DATA statements generated in Exercise 2b into a BASIC program.
    d. Loading the figure into a safe area of memory and using the memory move given in Listing 7-6.

8

Hi-Res Graphics

Objectives

After reading Chapter 8 you should be able to:

- Calculate the address for any byte on the Hi-Res screen.
- Calculate the value needed to produce a given dot pattern in a byte, and use both BASIC and Monitor to place that value in Hi-Res screen memory.
- Design, digitize, and display simple figures in Hi-Res.

A large percentage of the commercial software written for personal computers today uses Hi-Res graphics. Games, graphing packages, drafting design programs, several word processors, and even some of the alternate languages (for example, PILOT and LOGO) use the Hi-Res graphics screen for video output. Reflecting the pre-eminence of Hi-Res, the remainder of this book will be almost entirely devoted to various methods of controlling this mode of output.

Hi-Res gains an advantage over Low-Res because it lets you have individual control of each dot on the screen; there are 280 such dots horizontally across the screen and up to 192 dots vertically. The appearance of an image on the video screen is greatly affected by the degree of resolution (the number of dots per inch) in which it is drawn. In Chapter 6 you saw how the figure of the apple suffered when it was blocked out for Low-Res which has only four "dots" per inch. The Hi-Res dots are seven times smaller than the Low-Res blocks, so the Hi-Res mode produces figures that look much better; but as with any improvement, there is a price to pay.

Hi-Res adds another level to the addressing difficulty experienced with the non-contiguous Low-Res mode, and requires four times the memory for display. Add to that a reduced number of colors readily available to the programmer, plus the limitation of not being able to put all of the colors at any one location on the screen, and you may begin to wish you had never even heard of Hi-Res.

Fortunately, there are several Hi-Res graphics editors on the market which take most of the work and frustration out of creating Hi-Res figures. If you do not have one of these editors, we suggest that you acquire one before attempting any serious endeavor. A good Hi-Res editor will take care of the multitude of grubby details involved with Hi-Res graphics. It will let you draw lines, fill regions with color, and turn individual screen dots on and off. It will also let you save Hi-Res figures in shape tables (see Chapter 10), and even save the entire screen. Yet, even with an editor you must still understand the ins and outs of Hi-Res graphics.

In the following chapters we intend to show you a number of ways to tame the Hi-Res beast. In this chapter we will concentrate on mapping the Hi-Res maze, gaining control of individual dots, and creating simple figures in black and white. Color, animation, and other nifty topics will be discussed in later chapters.

Hi-Res Memory

There are two storage locations for Hi-Res graphics. The primary page, page 1, is located from 8192 through 16383 ($2000-$3FFF), and the secondary page, page 2, runs from 16384 through 24575 ($4000-$5FFF). Most of our work will be done with page 1, but the techniques and programs discussed will work identically on page 2 by simply adding 8192 ($2000) to the page 1 addresses.

Perhaps we should begin with a program designed to turn on every Hi-Res dot, one at a time and sequentially. We will start with location 8192 and turn on the first bit in that byte, then turn it off and turn on the second bit in the same byte. Then the second bit will go off and the third will go on, continuing to the eighth bit, which is left on when we begin with the next byte (location 8193) and do the same thing with it, and so forth throughout the entire screen. Even though we want to turn only one bit on or off at a time, we must POKE the entire byte, since bits do not have an address and hence cannot be referenced. For each byte we must POKE eight values, one after another, and each value will have only one bit on. The following lines of BASIC code will generate the eight values required.

```
FOR J = 1 TO 8
X = 2^(8 - J)
NEXT J
```

When J equals 1, X equals 2^7 equals 128, J equals 2 gives X equals 2^6 equals 64, and so on as indicated in Figure 8-1.

J	2^(8-J)	X	BINARY	HEX
1	2^7	128	1ØØØ ØØØØ	$8Ø
2	2^6	64	Ø1ØØ ØØØØ	$4Ø
3	2^5	32	ØØ1Ø ØØØØ	$2Ø
4	2^4	16	ØØØ1 ØØØØ	$1Ø
5	2^3	8	ØØØØ 1ØØØ	$Ø8
6	2^2	4	ØØØØ Ø1ØØ	$Ø4
7	2^1	2	ØØØØ ØØ1Ø	$Ø2
8	2^0	1	ØØØØ ØØØ1	$Ø1

Figure 8-1. Powers of two.

As promised, each value has only one bit turned on, and notice also that, as the index "J" goes from one to eight, the "on" bit in the byte goes from left to right. The following program uses this algorithm to successively POKE each value into a byte of Hi-Res memory.

```
1Ø REM TURN ON
2Ø REM HI-RES DOTS
3Ø REM
4Ø HGR : REM TURN HI-RES ON
5Ø X = PEEK (49234): REM SWITCH TO
   FULL PAGE
6Ø REM
7Ø FOR I = 8192 TO 16383
8Ø REM BEGINNING AND END OF HI-RES
9Ø FOR J = 1 TO 8
1ØØ X = 2 ^ (8-J): REM TURN ON BIT
11Ø POKE I,X: REM TURN ON DOT
12Ø FOR L = 1 TO 2: REM DELAY
13Ø NEXT L
14Ø NEXT J
15Ø NEXT I
16Ø CALL 65338: REM BEEP
17Ø END
```

Listing 8-1. Turn on Hi-Res dots.

From our discussion so far, you should expect a single dot to appear in the top left corner, and rapidly be replaced by one a little further to the right. That dot should also be quickly replaced by a third yet a little further right, and so on until the eighth dot, which remains on. The ninth dot should then begin just to the right of the still lit eighth dot, only to be replaced by the tenth still slightly more to the right, and so on; it should give the

impression of a dot travelling steadily to the right while leaving every eighth dot on.

Run the program and if the screen behaves in an unexpected—bordering on bizarre—manner, then the program is probably working perfectly.

Seeing Is Believing

What you should really see is a dot that appears close to the upper left hand corner of the screen, and then travels rapidly to the LEFT. Just as the first dot bumps into the edge of the screen, a second dot appears; it also moves to the left and stops where the first dot began. A third dot appears and moves left, and then a fourth and fifth and so on until the entire screen fills with a grid of little dots, all about a quarter-inch apart. (It requires approximately five minutes to reach this point.) The routine continues with the next set of dots appearing immediately beneath the first set. If you let the process continue long enough, the screen will finally display forty vertical lines of dots.

The display is mildly amusing, but after the novelty wears off you might realize that the "on" dots moved from right to left while the "on" bits in the byte moved from left to right. Apple Hi-Res displays the bits of a byte in the reverse order, so the right hand bit turns on the left hand dot of a byte, and the left hand bit turns on the right hand dot—er, sort of. You see, there is a little more to it.

Slow down the display process by changing the delay value in line 120 from 2 to a higher number. If you substitute a large enough value, you will be able to count the individual dots as they migrate to the left, and when you do, you will count only seven dots per byte to verify yet another Hi-Res oddity: the eighth (left-most) bit of the byte is not displayed at all. In other words, seven of the eight bits in each byte are displayed in reverse order, and the last (left-most) bit does not correspond to any dot at all. Although the reverse display of only seven bits may seem to be a capricious flight of "Wozniak whimsy" (named after Apple's designer), there truly are good reasons for that particular design. The left hand bit is used in a very clever scheme which facilitates Hi-Res color control.

While running the program you must have also noted that, as with Low-Res, the Hi-Res graphics screen fills in three distinct sections, and that there is a pause after filling each line in the lower section while the eight bytes of "phantom memory" are filled. By now those peculiarities should seem like familiar landmarks.

With all of these facts presented to you at once, you are bound to be a little confused. But one additional point should be made: with a little practice, Apple II Hi-Res is surprisingly powerful and easy to use. Suppose we demonstrate by turning on a few individual dots "manually."

Going Dotty

Each bit pattern creates a corresponding dot pattern on the screen. Since there are seven bits displayed from any byte, there are 128 possible combinations, and though we do not propose to show you every combination, Figure 8–2 lists which dots are turned on by which bits, and which ones may be handy for reference. You can see again that no dot corresponds to the left-hand bit.

```
1ØØØØØØØ    ! OFF  OFF  OFF  OFF  OFF  OFF  OFF
Ø1ØØØØØØ    ! OFF  OFF  OFF  OFF  OFF  OFF  ON
ØØ1ØØØØØ    ! OFF  OFF  OFF  OFF  OFF  ON   OFF
ØØØ1ØØØØ    ! OFF  OFF  OFF  OFF  ON   OFF  OFF
ØØØØ1ØØØ    ! OFF  OFF  OFF  ON   OFF  OFF  OFF
ØØØØØ1ØØ    ! OFF  OFF  ON   OFF  OFF  OFF  OFF
ØØØØØØ1Ø    ! OFF  ON   OF   OFF  OFF  OFF  OFF
ØØØØØØØ1    ! ON   OFF  OFF  OFF  OFF  OFF  OFF
```

Figure 8–2. Bit correspondence.

To turn on a dot, you must also know the address of the byte which controls it. Refer to the Hi-Res memory map in Figure 8–3.

The boxes shown in the map are very similar to the ones used in the Text/Low-Res map in Chapter 6, and the address of each box is found in the same manner—add the addresses given on the map for the row and column of the box. Within each box there are eight rows of seven dots stacked vertically as shown in the inset, and each row of dots in the box contributes an additional number called the position address. To turn on a dot, you must first find the address of the box and the row of dots within that box which contains the dot you are looking for. If you are not confused by now, then you must have worked with this before! Perhaps we can clear things up by use of an example.

Example 1

Suppose that you wish to display the seventeenth dot from the left in the third row of dots.

Solution

The third row of dots is within the first row of boxes, and since each box displays seven (not eight) dots across, the seventeenth dot is in the third box across as shown in Figure 8–4. Now that you have located the dot, you need to add three numbers to get its address: the address for the box's row, column, and the byte's position within the box.

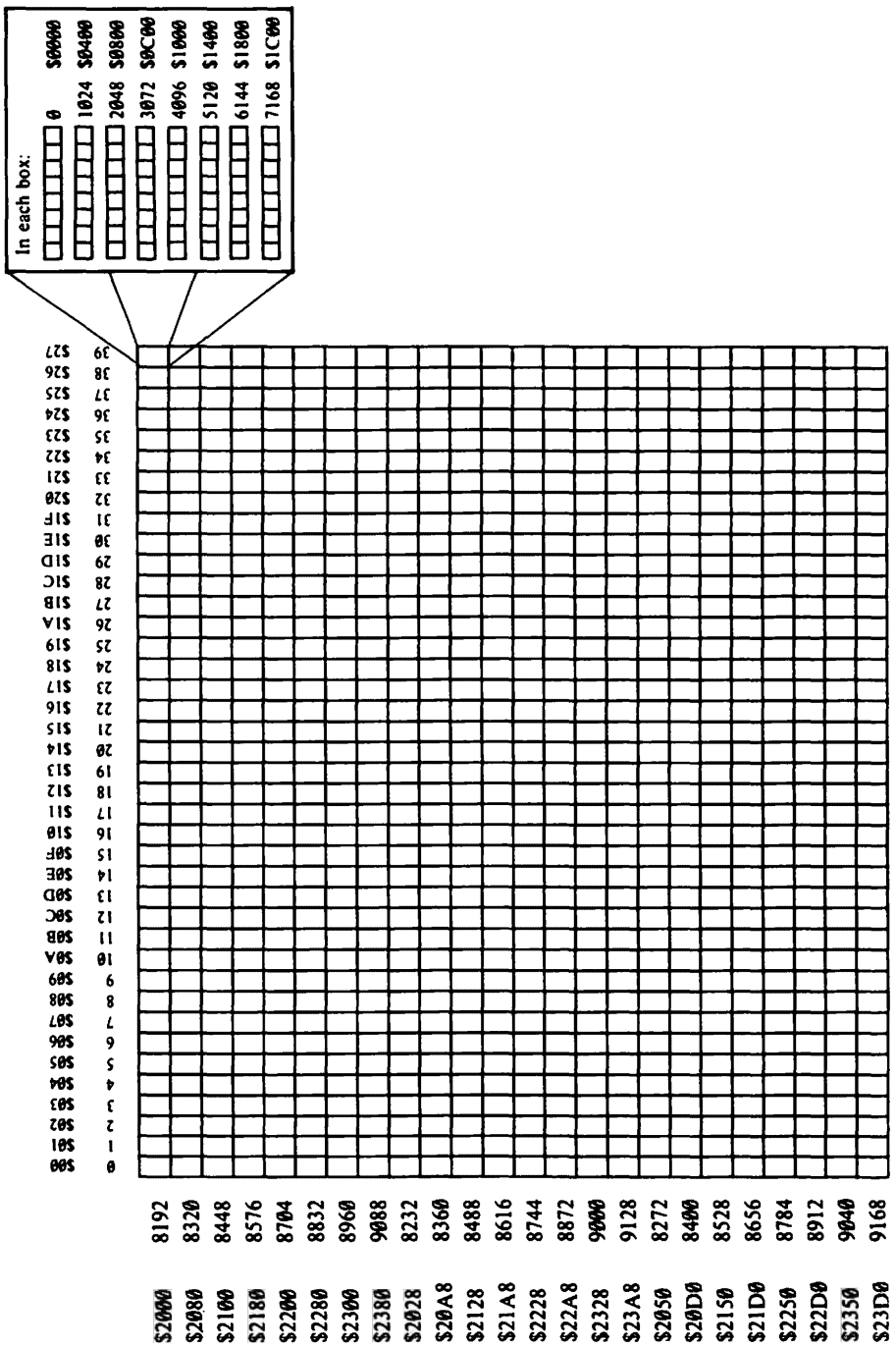


Figure 8–3. Hi-Res memory map.

(Used with permission of Apple Computer Corporation)

To obtain the address for any byte, add the addresses for that byte's box row, box column, and position in box.

```
Row Address          8192      $2ØØØ
Column Address       2         $  2
Position Address     2Ø48      $Ø8ØØ
                     -----     -----
Final Address        1Ø242     $28Ø2
```

Thus, you at last arrive at the address, but now you need to calculate the value to put into it. To do that, again refer to Figure 8–4.

The top line of the figure shows the first three bytes of screen memory, taking into account that only seven bits of each byte are actually displayed. Dot number seventeen is the third dot from the left in its box; so to turn it on, you would turn on the third bit from the RIGHT in the byte as shown in the bit pattern on the second line. Remember, the bit pattern is the reverse of the dot pattern.

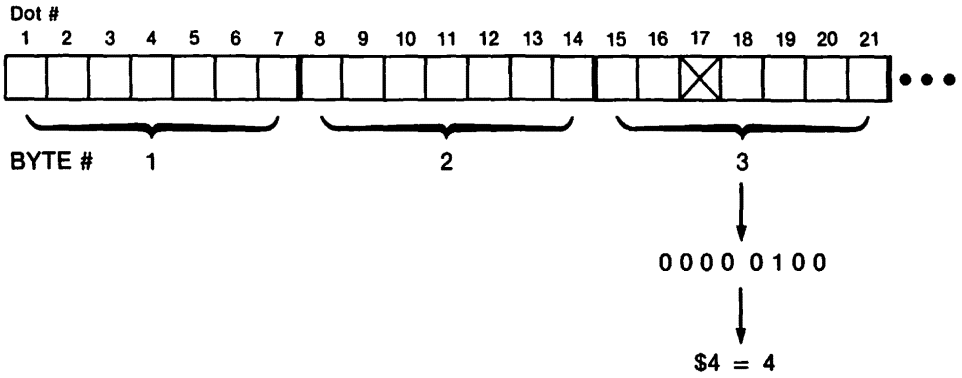


Figure 8–4. Display conversion.

The left-hand bit has been shown in the second line as a zero (off). The binary number in the second line is converted to hex and decimal form in the third line, and the decimal value is the one we are about to POKE. After all that work, you can turn on the desired dot using the following commands from BASIC.

```
HGR                  (set graphics)
POKE 1Ø242,4         (turn on the dot)
```

You could use Monitor by typing

```
HGR
CALL -151
28Ø2: 4
```

Example 2

Turn on dots 57, 58, and 59 in line number 116 of the Hi-Res screen.

Solution

If you count eight lines per box, the fourteenth row of boxes ends with the 112th line of dots (8*14=112), so the 116th line is the fourth line in the fifteenth box. Therefore, by using the memory map in Figure 8–3, the row address is 9000, and the position in the box address is 3072. Dots 57, 58, and 59 of that line are the second, third, and fourth dots in the eighth box, which has column address 7 ($7), so the final address is:

```
9000        $2328
3072        $0C00
   7        $  7
-----       -----
12079       $2F2F
```

The display for that byte would look like:

```
-XXX---
```

where - represents a dot which is off, and X a dot which is on. Reversing the order and supplying the "missing" left zero would give you the bit pattern: 00001110, which is equal to 14 ($0E).

Now that you have determined the address (12079 or $2F2F) and the value (14 or $0E), type:

```
 HGR
 POKE 12079, 14
```

or

```
 HGR
 CALL -151
 2F2F: E
```

This is not so bad after you get used to it, is it? Perhaps one more example ....

Example 3

Turn on the dots 27 and 30 in line 16 of the Hi-Res screen.

Solution

Line 16 is the last byte in the second row of boxes, so the row address is 8320 ($2080) and the position address is 7168 ($1C00). Dots 27 and 30 are in separate boxes—the fourth and fifth columns—so they will have different column addresses, and we will have to POKE each byte separately.

The fourth column address is 3 ($3), so the address of the first byte we are after is 8320 + 7168 + 3 = 15491 ($2080 + $1C00 + $3 = $3C83), and the address of the second byte is 15492 ($3C83). The dot patterns for the fourth and fifth byte look like:

```
-----X- -X-----
```

so the bit patterns would be:

```
010 0000 and  000 0010
```

and when you supply the left hand zero you get:

```
0010 0000   and 0000 0010
=  $20              $ 2
=  32                2
```

Hence, to plot the dots, type:

```
HGR
POKE 15491,32
POKE 15492,2
```

If you were to supply ones for the left-hand bits you would get:

```
1010 0000  and   1000 0010
=  $A0            $82
=  160            130
```

so:

```
HGR
POKE 15491,160
POKE 15492,130
```

Notice that the same dots were turned on with both sets of statements, but that they were different colors—not very impressive unless you have a color display, but it does illustrate how the eighth bit controls color in Hi-Res.

Monitor may be used to turn on the same dots, and also to clear the screen without the use of the HGR statement.

The Hi-Res screen may be cleared by moving zeros into every byte, so type:

```
CALL -151
2000:0
2001<2000.3FFEM
3C83:60 03
```

The first command invokes Monitor, the next two clear the screen (see Chapter 3), and the last places the proper byte in each of the two addresses.

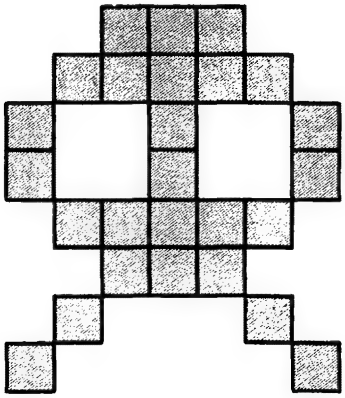


Figure 8-5. Alien critter.

(From The Artist by Warren Schwader, used with permission.)

Whew! That seems like a lot of work just for a few dots!! As it happens, working up an entire figure is not much harder. However, your preliminary planning is extremely important when designing a figure in Hi-Res—let's put together an alien critter as an example. Figure 8-5 shows a sample alien already drawn on graph paper. For simplicity, it is only seven dots wide and eight rows high so that it will fit within a single box of the memory map.

The first step is to calculate the value of each byte composing the alien. Figure 8-6 shows the dot patterns, the required bit patterns, and the corresponding hex and decimal values.

DOT PATTERN	BIT PATTERN	HEX VALUE	DEC VALUE
--XXX--	ØØØ1 11ØØ	$1C	28
-XXXXX-	ØØ11 111Ø	$3D	61
X--X--X	Ø1ØØ 1ØØ1	$49	73
X--X--X	Ø1ØØ 1ØØ1	$49	73
-XXXXX-	ØØ11 111Ø	$3D	61
--XXX--	ØØØ1 11ØØ	$1C	28
-X---X-	ØØ1Ø ØØ1Ø	$22	34
X-----X	Ø1ØØ ØØØ1	$41	65

Figure 8-6. Alien translation.

The extra zero has been placed to the left of each bit pattern to supply the "missing" eighth bit. Calculating the values was not difficult at all. Now you need to calculate the addresses.

The row and column addresses depend on which box you will use to plot the alien, but the address for the position in the box is fixed by the row's position in the figure. The position address for row one is 0 ($0), for row two it is 1024 ($400), for row three it is 2048 ($800), and so forth, as shown in the Hi-Res memory map (see Figure 8–3). The following program takes advantage of those facts as it plots the alien in any box you choose on the mixed mode Hi-Res screen.

```
10 REM ALIEN PLOT
20 REM
30 REM INITIALIZATION
40 DIM RA%(20)
50 FOR I = 1 TO 20: READ RA%(I):
   NEXT I
60 REM
70 REM INPUT ROW AND COLUMN
80 HGR : HOME : VTAB (24)
90 INPUT "ROW NUMBER? (1-20)";R$
100 R% = VAL (R$)
110 IF R% < 1 OR R% > 20 THEN PRINT
    CHR$ (7): GOTO 90
120 INPUT "COLUMN NUMBER? (1-40)";C$
130 C% = VAL (C$)
140 IF C% < 1 OR C% > 40 THEN PRINT
    CHR$ (7): GOTO 120
150 BA% = RA%(R%) + C%-1
160 POKE BA%,28
170 POKE BA% + 1024,62
180 POKE BA% + 2048,73
190 POKE BA% + 3072,73
200 POKE BA% + 4096,62
210 POKE BA% + 5120,28
220 POKE BA% + 6144,34
230 POKE BA% + 7168,65
240 REM
250 PRINT "PLOT ANOTHER? (Y/N)": GET A$
260 IF A$ < > "N" GOTO 90
270 TEXT : HOME
280 END
290 DATA 8192,8320,8448,8576,8704,
    8832,8960,9088,8232,8360
300 DATA 8488,8616,8744,8872,9000,
    9128,8272,8400,8528,8656
```

Listing 8-2. Alien plot.

Lines 40 through 60 dimension and initialize an array containing the twenty row addresses. Line 80 sets the mixed Hi-Res graphics mode with

the cursor at the bottom of the text window. Lines 90 through 150 accept your input for the row and column numbers and, after checking to insure they are valid, converts them to a base address for the box indicated. Line 150 calculates the base address by adding the row address for the box [RA%(R%)] to the column address (which happens to be one less than the column number). Having added the row and column addresses for the box, it only remains to add the address for the position in the box for each byte of the alien (how do aliens taste?). The final address is calculated and the proper value POKEd into it in lines 160 through 230, lines 250 and 260 give you a chance to plot another one, and then lines 270 and 280 wrap up the program. The data in lines 290 and 300 are the row addresses read in line 50. The plotting speed may be enhanced somewhat by calculating all of the final addresses before starting to POKE any of them.

Example 4

Write a simple BASIC program to plot the cannon given in Figure 8–7.

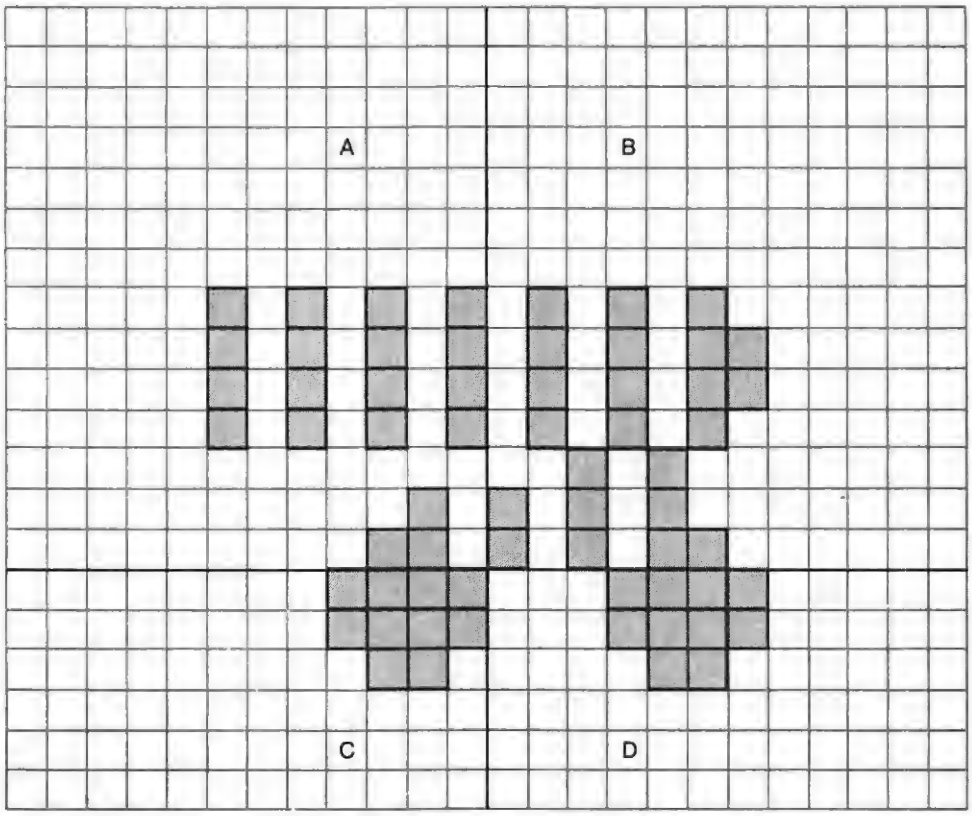


Figure 8–7. Hi-Res cannon.
(From Cannonball Blitz by Olaf Lubek, with permission.)

Solution

The first task is to digitize the cannon. Notice that it is fourteen dots wide and ten bytes high, so it will require four boxes to display. The figure has already been divided into four quadrants. Figures 8-8A-D show the digitizing of the corresponding quadrant.

DOT PATTERN	BIT PATTERN	HEX VALUE	DEC VALUE
X-X-X-X	1Ø1 Ø1Ø1	$55	85
X-X-X-X	1Ø1 Ø1Ø1	$55	85
X-X-X-X	1Ø1 Ø1Ø1	$55	85
X-X-X-X	1Ø1 Ø1Ø1	$55	85
-------	ØØØ ØØØØ	$ØØ	Ø
-----X-	Ø1Ø ØØØØ	$2Ø	32
----XX-	Ø11 ØØØØ	$3Ø	48
---XXXX	111 1ØØØ	$78	12Ø

Figure 8-8A.

DOT PATTERN	BIT PATTERN	HEX VALUE	DEC VALUE
-X-X-X-	Ø1Ø 1Ø1Ø	$2A	42
-X-X-XX	11Ø 1Ø1Ø	$6A	1Ø6
-X-X-XX	11Ø 1Ø1Ø	$6A	1Ø6
-X-X-X-	Ø1Ø 1Ø1Ø	$2A	42
--X-X--	ØØ1 Ø1ØØ	$14	2Ø
X-X-X--	ØØ1 Ø1Ø1	$15	21
X-X-XX-	Ø11 Ø1Ø1	$35	53
---XXXX	111 1ØØØ	$78	12Ø

Figure 8-8B.

DOT PATTERN	BIT PATTERN	HEX VALUE	DEC VALUE
---XXXX	111 1ØØØ	$78	12Ø
----XX-	Ø11 ØØØØ	$3Ø	48

Figure 8-8C.

DOT PATTERN	BIT PATTERN	HEX VALUE	DEC VALUE
--XXXX	111 1ØØØ	$78	12Ø
----XX-	Ø11 ØØØØ	$3Ø	48

Figure 8-8D.

The following program will POKE into memory the values we just generated. We will plot the cannon in the upper left corner of the screen.

```
10 REM POKE CANNON
20 REM THE HARD WAY!
30 REM
40 HGR
50 REM QUADRANT A
60 POKE 8192,85: POKE 9216,85:
   POKE 10240,85: POKE 11264,85
70 POKE 12288,0: POKE 13312,32:
   POKE 14336,48: POKE 15360,120
80 REM QUADRANT B
90 POKE 8193,42: POKE 9217,106:
   POKE 10241,106: POKE 11265,42
100 POKE 12289,20: POKE 13313,21:
    POKE 14337,53: POKE 15361,120
110 REM QUADRANT C
120 POKE 8320,120: POKE 9344,48
130 REM QUADRANT D
140 POKE 8321,120: POKE 9345,48
150 END
```

Listing 8-3. Cannon poker.

Listing 8-3 makes no pretense at cleverness, but it does perform as promised.

By now you have seen how the individual bits on the Hi-Res screen are controlled, and how you can wade through the memory map to find the byte of your dreams. Using those ideas, you have developed the data for and plotted some simple figures on the screen. Not bad for a day's work, eh?

Vocabulary

Column Address

Digitizing

Position Address

Row Address

Exercises

1. Place the given dot pattern where indicated on the Hi-Res screen.
   a. -XX-X-X in row 10, column 20 of boxes, and first byte of the box.

b. XX-XX-- in the fifth line and eleventh column of dots.

c. X-----X in the fourteenth line and 27th column of dots.

2. Write a BASIC program which lets the user plot the cannon from Example 3 in any group of four boxes on the Hi-Res screen.

3. Digitize and display the butterfly figure given below.

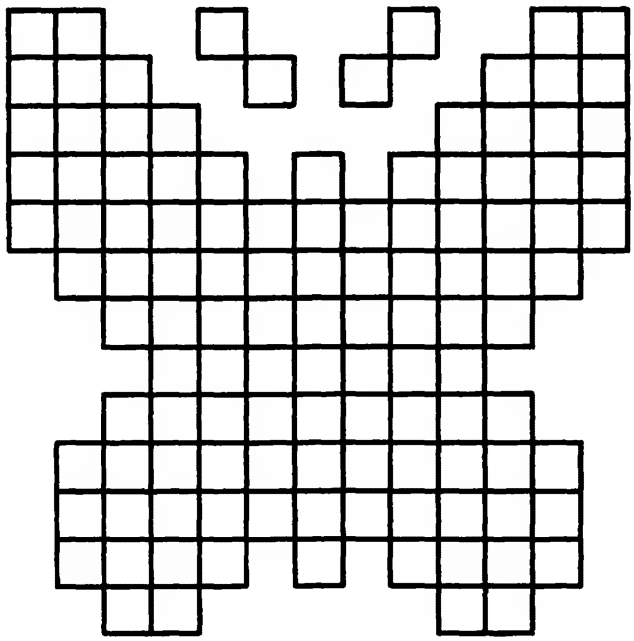


Figure 8-9. A butterfly.

9

Hi-Res Color

Objectives

After reading Chapter 9 you should be able to:

- Design and display a Hi-Res color figure.
- When given the location of a dot on the screen and a desired color, determine if that color can be displayed at that location, and if so, calculate the byte of memory which contains that dot and POKE the appropriate value into that location.
- Use dithering to generate colors other than the six basic Hi-Res shades.

Good color is virtually a requirement for quality Hi-Res products; most software publishers will not touch a Hi-Res program unless its color effects are clear, clean, and smooth. Color enhances the display by distinguishing the different elements of a picture or graph, by adding life to a screen, and by making the output more attractive to the eye of the viewer. Your Apple's hardware is designed to produce six colors, including black and white and, by a software process known as **dithering,** can be made to display upwards of twenty additional hues.

Getting color on the Hi-Res screen is simple—almost too simple since any time you plot a single dot, it is always in color. The trick is to develop the right color in the right place. By now you should be familiar with the way Hi-Res memory is laid out, and how to use the memory map to find your way around. You also need some facility in translating desired dot patterns into their corresponding bit patterns, and then into their hex or decimal values. Another necessary item is either a color display, or a very vivid imagination.

A Bit of Color

In Chapter 8, we pointed out that the left-most bit in a Hi-Res byte was not displayed, and alluded to its function in color control. That bit is called the **color bit** of the byte, and its setting determines which group of colors the remaining seven bits will display. If the color bit of a byte is off (zero), the remaining seven bits will display colors from group 1: BLACK1, GREEN, VIOLET, or WHITE1. If the color bit is on (one), the byte will display group 2 colors: BLACK2, ORANGE, BLUE, or WHITE2. Those colors will vary depending on the TV or monitor used, but you can usually adjust the tint on your screen so the proper colors are displayed. We will return to the color bits and color groups; meanwhile, run the following program as an experiment.

```
10 HGR
20 HCOLOR = 3
30 HPLOT 1,0 TO 1,100
40 GET P$: REM PAUSE
50 HCOLOR = 4
60 FOR I=0 TO 100
70 HPLOT 0,I
80 NEXT I
```

Listing 9-1. Vertical line.

According to the Apple manuals, HCOLOR = 3 is WHITE1 (the white belonging to color group 1); hence, lines 10 through 30 should plot a vertical white line in column 1. But when you run the program you are instead greeted by a vertical green line! That is because you are plotting a single dot in each row, and any isolated dot always results in a color. But there is more. Line 40 halts the program until you press a key, line 50 sets the color to BLACK2, which is the black belonging to color group 2. Lines 60, 70, and 80 plot a vertical line of black dots in column 0. Since column 0 was already black, you would not expect anything to change, and yet the green line slowly changes to orange after you press a key.

The reason for the color change has to do with the color bit. The dots in column 0 and column 1 are controlled by the same byte of Hi-Res memory, and plotting WHITE1 turned on the column 1 bit of each byte and left the rest of the bits off, including the color bit. Plotting column 0 in BLACK2 turned off the column 0 bits, which were off anyway, but also turned on the color bit of each byte since BLACK2 is a group 2 color. When the color bit of each byte was turned on, it caused the entire byte to display group 2 colors; thus, the green in column 1, which is a group 1 color, changed to the corresponding group 2 color, orange. If column 0 had been plotted in BLACK1 instead of BLACK2, the color bit of each byte would have remained off and no color change would have occurred.

Let's run a program which illustrates yet another characteristic of Hi-Res color.

```
10 HGR
20 HCOLOR = 2
30 FOR I = 1 TO 100
40 HPLOT I,0
50 NEXT I
60 REM
70 REM NOW TURN HALF OFF
80 HCOLOR = 0
90 FOR I = 2 TO 100 STEP 2
100 HPLOT I,0
110 NEXT I
120 END
```

Listing 9-2. Horizontal line.

An examination of the listing would lead one to believe that the program will plot 100 consecutive horizontal blue dots, and then turn all the even dots off. However, when the program is run, the blue line is drawn, and then erased completely! How can that be when you plot 100 dots and only erase 50? If you care to insert a delay loop and signal in the program `35 FOR J = 1 TO 500: NEXT J: PRINT CHR$(7)` and count the dots as they are plotted, you will find that only 50 dots are plotted in the first place. Blue is only available in even numbered columns (0,2,4,...), and so when the program attempts to plot blue on the odd numbered dots, it has no effect at all. Violet is also available on only even dots, while odd dots may display either green or orange.

To summarize what you have seen so far, if a dot is plotted in an even numbered column, it will be either violet or blue, and if a dot is in an odd column, it will be green or orange. The color bit selects which of the available colors each dot will display: if the color bit is off (group 1), even dots display violet and odd dots display green; if the color bit is on (group 2) then even dots are blue and odd dots are orange. The left-most column of dots on the screen is counted as column zero, and columns 1 through 279 follow sequentially across the screen.

Black is plotted by turning a dot off, and white is plotted by turning on two adjacent dots in the same row. Let's experiment using the POKE statement. Type:

```
HGR
POKE 8192,1
POKE 8192,2
POKE 8192,3
```

In the upper left corner of the Hi-Res screen you will display a single violet dot, then a lone green dot, and finally a white dot. How did the white dot

get there? The chart below takes the values that you POKEd and shows the corresponding bit patterns and effects.

VALUE	BIT PATTERN	DOT(S) ON
1	ØØØØ ØØØ1	Row Ø
2	ØØØØ ØØ1Ø	Row 1
3	ØØØØ ØØ11	Row Ø and Row 1

Figure 9-1. Bit patterns and effects.

When the dots were turned on singly, they displayed their respective colors, but when they were both on they displayed white. Any "on" dot next to another "on" dot will display white. Plotting one dot under or over another has no effect on the color.

How Do You Like Them Apples?

To practice with Hi-Res colors, we will plot the apple given in Figure 9-2. The figure has already been sectioned into the nine boxes required to display it, and Figures 9-2A through 9-2I show the conversion from dot patterns to decimal values for each of the sections.

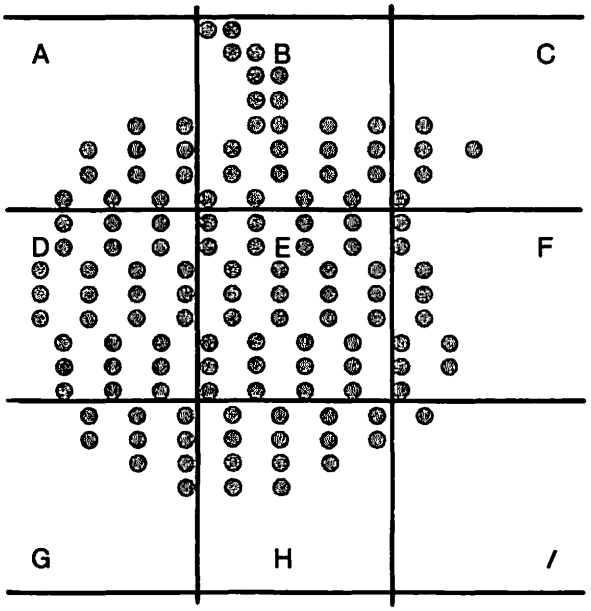


Figure 9-2. Hi-Res apple.

Except for the stem, which will be white, the grid for the apple is composed of dots in every second column. That pattern is necessary since

each Hi-Res color is available on only odd or only even dots, and further, if two adjacent dots are turned on, they will both display white.

Colors from both groups are used as indicated in Figure 9-2.

DOT PATTERN	BIT PATTERN	HEX	DEC
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
----X-X	ØlØl ØØØØ	$5Ø	8Ø
--X-X-X	ØlØl ØlØØ	$54	84
--X-X-X	ØlØl ØlØØ	$54	84
-X-X-X-	ØØlØ lØlØ	$2A	42

Figure 9-2A.

DOT PATTERN	BIT PATTERN	HEX	DEC
XX-----	ØØØØ ØØ11	$Ø3	3
-XX----	ØØØØ Ø11Ø	$Ø6	6
--XX---	ØØØØ 11ØØ	$ØC	12
--XX---	ØØØØ 11ØØ	$ØC	12
--XX---	ØØØØ 11ØØ	$ØC	12
-X-X-X-	ØØlØ lØlØ	$2A	42
-X-X-X-	ØØlØ lØlØ	$2A	42
X-X-X-X	ØlØl ØlØl	$55	85

Figure 9-2B.

DOT PATTERN	BIT PATTERN	HEX	DEC
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
X-X----	ØØØØ ØlØl	$Ø5	5
X-X-X--	ØØØl ØlØl	$15	21
X-X----	ØØØØ ØlØl	$Ø5	5
-X-----	ØØØØ ØØlØ	$Ø2	2

Figure 9-2C.

DOT PATTERN	BIT PATTERN	HEX	DEC
-X-X-X-	ØØ1Ø 1Ø1Ø	$2A	42
-X-X-X-	ØØ1Ø 1Ø1Ø	$2A	42
X-X-X-X	11Ø1 Ø1Ø1	$D5	213
X-X-X-X	11Ø1 Ø1Ø1	$D5	213
X-X-X-X	11Ø1 Ø1Ø1	$D5	213
-X-X-X-	1Ø1Ø 1Ø1Ø	$AA	17Ø
-X-X-X-	1Ø1Ø 1Ø1Ø	$AA	17Ø
-X-X-X-	1Ø1Ø 1Ø1Ø	$AA	17Ø

Figure 9-2D.

DOT PATTERN	BIT PATTERN	HEX	DEC
X-X-X-X	Ø1Ø1 Ø1Ø1	$55	85
X-X-X-X	Ø1Ø1 Ø1Ø1	$55	85
-X-X-X-	1Ø1Ø 1Ø1Ø	$AA	17Ø
-X-X-X-	1Ø1Ø 1Ø1Ø	$AA	17Ø
-X-X-X-	1Ø1Ø 1Ø1Ø	$AA	17Ø
X-X-X-X	11Ø1 Ø1Ø1	$D5	213
X-X-X-X	11Ø1 Ø1Ø1	$D5	213
X-X-X-X	11Ø1 Ø1Ø1	$D5	213

Figure 9-2E.

DOT PATTERN	BIT PATTERN	HEX	DEC
-X-----	ØØØØ ØØ1Ø	$Ø2	2
-X-----	ØØØØ ØØ1Ø	$Ø2	2
X-X----	1ØØØ Ø1Ø1	$85	133
X-X----	1ØØØ Ø1Ø1	$85	133
X-X-X--	1ØØ1 Ø1Ø1	$95	149
-X-X---	1ØØØ 1Ø1Ø	$8A	138
-X-X---	1ØØØ 1Ø1Ø	$8A	138
-X-----	1ØØØ ØØ1Ø	$82	13Ø

Figure 9-2F.

DOT PATTERN	BIT PATTERN	HEX	DEC
--X-X-X	Ø1Ø1 Ø1ØØ	$54	84
--X-X-X	Ø1Ø1 Ø1ØØ	$54	84
----X-X	Ø1Ø1 ØØØØ	$5Ø	8Ø
------X	Ø1ØØ ØØØØ	$4Ø	64
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø

Figure 9-2G.

DOT PATTERN	BIT PATTERN	HEX	DEC
-X-X-X-	ØØ1Ø 1Ø1Ø	$2A	42
-X-X-X-	ØØ1Ø 1Ø1Ø	$2A	42
-X-X-X-	ØØ1Ø 1Ø1Ø	$2A	42
-X-X---	ØØØØ 1Ø1Ø	$ØA	1Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø

Figure 9-2H.

DOT PATTERN	BIT PATTERN	HEX	DEC
X-X----	ØØØØ Ø1Ø1	$Ø5	5
X------	ØØØØ ØØØ1	$Ø1	1
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø
-------	ØØØØ ØØØØ	$ØØ	Ø

Figure 9-2I.

In Figures 9-2D through 9-2F, some of the color bits are turned on in order to select group 2 colors.

Now that you have digitized the figure, you need a program such as Listing 9-3 below to place those values in Hi-Res screen memory. The program lets you select the position of the apple on the screen, within limits. You will be asked for the row and column number for the upper left corner of the apple (Figure 9-2A); the rest of the apple follows from there.

```
 10 REM COLOR APPLE
 20 REM
 30 REM
 40 REM INITIALIZE
 50 REM
 60 OFST% = 1024: DIM SA%(20)
 70 FOR I = 1 TO 20: READ SA%(I): NEXT
 80 HGR : VTAB (23)
 90 REM
100 REM GET ROW & COL #'S
110 REM
120 INPUT "BOX ROW #? (1-18)";R$
130 R% = VAL (R$): IF R% < 1 OR
    R% > 18 THEN PRINT CHR$ (7):
    GOTO 120
140 INPUT "BOX COL. #? (1-38)";C$
150 C% = VAL (C$): IF C% < 1 OR
    C% > 38 THEN PRINT CHR$ (7):
    GOTO 140
160 REM
170 REM PLOT APPLE
180 REM
190 FOR K = 0 TO 2
200 FOR J = 0 TO 2
210 BA% = SA%(R% + K) + (C%-1) + J
220 FOR I = 1 TO 8
230 READ V%: POKE BA% + (I-1) *
    OFST%,V%
240 NEXT I,J,K
250 END
370 REM
380 REM ADDR. TABLE
390 REM
400 DATA 8192,8320,8448,8576,8704,
    8832,8960,9088,8232,8360,8488
410 DATA 8616,8744,8872,9000,9128,
    8272,8400,8528,8656
480 REM
490 REM  APPLE TABLE
```

```
500 REM
510 DATA 0,0,0,0,80,84,84,42
520 DATA 3,6,12,12,12,42,42,85
530 DATA 0,0,0,0,5,21,5,2
540 DATA 42,42,213,213,213,170,170,170
550 DATA 85,85,170,170,170,213,213,213
560 DATA 2,2,133,133,149,138,138,130
570 DATA 84,84,80,64,0,0,0,0
580 DATA 42,42,42,10,0,0,0,0
590 DATA 5,1,0,0,0,0,0,0
```

Listing 9-3. POKE color apple.

Line 60 initializes the variable OFST%, which represents the offset or difference in addresses between each row of dots within a box. Line 60 also DIMensions the array SA%, which contains the starting addresses for the twenty rows of boxes available on the left side of the mixed screen. The variables are indicated as integers in order to speed execution. Line 70 READs the values for the starting address from the address data table, and line 80 sets the Hi-Res mode with the cursor in the text window.

Lines 120 and 130 input the row number and check to insure that it is within the proper range. Even though there are twenty rows of boxes on the mixed Hi-Res screen, the starting box for the apple must be between 1 and 18 since the figure occupies three boxes vertically, and beginning at line nineteen would cause the apple to try to plot in the range occupied by the text window. Even though it is possible to plot in the last four rows, the apple would not be visible in mixed mode, so no provision has been made to do so in this listing.

Lines 140 and 150 accept the column number and verify that it is between 1 and 38 so that the entire width of the apple will plot on the screen.

Lines 190 through 240 do the actual POKEing of the data values developed in Figures 9-2A through 9-2I. The variable "K" indexes the rows while "J" indexes the columns; they each take on three values, 0, 1, and 2. In line 210, the address of the box to be filled is calculated by adding the starting address of the row of boxes being filled (SA%(R% + K)), to the column address ((C%-1) + J).

Lines 220-240 read each of the eight data values for each box, calculate the final addresses by adding the proper multiple of the offset to the base address, and then POKE each value into its address.

Run the program a few times and try different locations for the apple. You will notice that the picture changes color depending on which column you select. To help understand why, let's consider the dots at the extreme left of the apple: the third, fourth, and fifth rows of box D. When that box is plotted on the left edge of the screen, those dots will display

blue as they are in an even column (zero) and a group 2 byte. When box D is plotted in the second column of boxes from the left, those same dots display orange since they are now in an odd column of dots (seven). That pattern alternates across the width of the screen.

Another weakness of the program is that you are constrained to always select the apple's position in increments of a full box. That is, you may not choose to draw the apple starting three and one-half boxes into the screen, but instead you must always use a whole number of boxes. There are techniques which let you POKE the figure beginning with any of the 280 columns and 160 rows on the screen, but they will not be pursued here, as the results are better accomplished using the methods discussed in the chapters covering Shapes and Byte-Move Graphics.

Everything You Know Is False

In this section we appear to contradict a portion of what we have just told you. We have said (repeatedly) that there are 280 positions across the Hi-Res screen, but now we attempt to convince you that there are only 40 positions, and also that there are 560 positions, and sometimes 140 positions. ARRGH!! Hopefully, by temporarily confusing the issue, we will clarify it. The purpose is to provide you with several clear and accurate models for use with the Hi-Res screen—not physical models, but rather conceptual ones which will help structure your thinking and simplify your task when designing Hi-Res output. Each model is based on a different number of positions across the screen, but they do use the same number of positions down the screen: 160 rows of dots (192 for full screen).

Why 40?

The explanation of the 40 position model is easiest since you have already dealt with it, perhaps unknowingly. To process any dot, you must unavoidably deal with the byte which contains that dot, and there are only 40 bytes across each line of the screen. When you developed and plotted the Hi-Res apple, your design was developed in one-byte increments and plotted with the consideration of having only 40 boxes or bytes across the screen in which to draw the figure.

What About 280?

The 280-position model is still valid. Each of the 40 bytes displays seven dots, and that yields 280 dots across the screen. The 280-dot concept is useful when you are designing your figures, but when you begin to digitize the data, you naturally adopt the 40-position model, perhaps without realizing it. Refer to the section where you worked with the Hi-Res apple, and

notice how the 280-dot model led into the 40-byte model with no trouble at all when you began to digitize the figure.

Only 140

When working with colors on the Hi-Res screen, most game programmers consider the width of the screen to be composed of only 140 available dots. Clearly, it requires two adjacent dots to display white, and if you would display a line of green dots, then between each green dot there is a black one, so you can see that it takes two dots to display a color also. If you look back at Figure 9-2, the picture of the Hi-Res apple, you will see that only every second dot was used for the colored portions. Since a unit of color requires two dots, you may reasonably interpret the screen to allow only 140 units of color across.

Further, when you plotted the apple in different positions, the colors changed depending on where it was plotted. In order to maintain the original colors, any dot that was originally even must always be plotted as an even dot, and the same idea follows for the odd dots. To accomplish that, the figure must be moved in multiples of two dots at a time, so for this reason also, there are only 140 possible positions across the screen.

Would You Believe 560?

Despite what the Apple documentation says, there are actually 560 dots available across the Hi-Res screen. By now you probably think we are totally bonkers, so we will prove our point by demonstration. First, enter the Monitor, and then the Hi-Res mode, by typing:

```
CALL -151
C050
C057
```

Next, clear the decks for action by typing:

```
2000: 0
2001<2000.3FFEM
```

If this seems obscure to you, perhaps another reading of the Monitor and Soft Switch chapters is needed.

Figure 9-3 shows the hex address of the first byte for each of the first 14 screen lines, and the dot and bit patterns for 14 values to be placed in those locations.

HEX ADDRESS	HEX VALUE	BIT PATTERN	DOT PATTERN
2000	01	0000 0001	X------
2400	81	1000 0001	X------
2800	02	0000 0010	-X-----
2C00	82	1000 0010	-X-----
3000	04	0000 0100	--X----
3400	84	1000 0100	--X----
3800	08	0000 1000	---X---
3C00	88	1000 1000	---X---
2080	10	0001 0000	----X--
2480	90	1001 0000	----X--
2880	20	0010 0000	-----X-
2C80	A0	1010 0000	-----X-
3080	40	0100 0000	------X
3480	C0	1100 0000	------X

Figure 9-3.

Place the values in memory by typing

```
2000: 01
2400: 81
2800: 02
  .
  .
  .
3480: C0
```

Looking at the dot patterns, the expected result is seven pairs of dots, with one dot of the pair directly above the other.

However, when you enter the values, what you get is a diagonal line; the pairs of dots are not stacked as we expected, even though the same dot was turned on in each byte of the pair. This phenomenon relies on the fact that the first byte in each pair was in color group one, and the second byte was from group two, as determined by the color bit on the left side of the byte.

One byte of screen memory actually controls fourteen dots on the screen, and the two sets of seven dots are interleaved as shown in Figure 9-4.

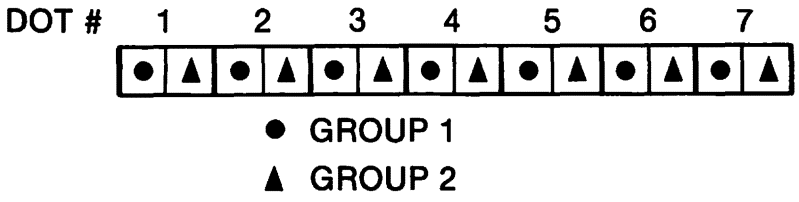


Figure 9-4. Double dot picture.

One set of seven dots may be displayed by setting the color bit to display group 1, and the other seven are displayed by setting the color bit to group 2. For example, change the group two value in location $2400, which is presently 1000 0001, to the equivalent group one value, 0000 0001, and then back again by typing:

```
2400: 01
2400: 81
```

You will see the dot in line two move slightly to the left, and then back to the right even though in both cases it was the left-most dot of the group being displayed.

The point is that there are actually 560 dots across the Hi-Res screen, though only half the dots in any byte are available at any one time. The fact that the group 2 dots are slightly to the right of the group 1 dots may be exploited to draw a smoother line than possible with just one color group, as we did in the example, or to smooth the outlines of figures on the screen in order to improve their looks.

How's That Again?

The result of all this discussion is that there are 560 dots across the Hi-Res screen, 280 of which are potentially displayable. There are 140 pairs of dots available when generating color, and all the data for a line on the screen must fit into 40 bytes. You use the units and model which are most convenient for your immediate purpose.

Hither, Dither, and Yon

According to the Apple documentation, there are only six colors available in Hi-Res mode, including black and white. Since Apple built the computer one would expect them to be correct, and they are—sort of. There are programs available in Hi-Res which use many more than six colors, and that seems to contradict the reference manuals.

As you have seen for yourself, the dots on the Hi-Res screen are truly capable of producing only black, white, violet, blue, green, and orange; but when you add to that a good measure of programmer imagination and creativity, it is possible to artificially produce extra hues by a process called "dithering."

Run the following program.

```
10 REM DITHER DEMO
20 REM
30 DLAY = 200: REM DELAY VALUE
40 SZ = 50: REM SIZE
```

```
5Ø HGR
6Ø FOR C1 = 1 TO 7
7Ø FOR C2 = 1 TO 7
8Ø FOR Y = Ø TO SZ STEP 2
9Ø HCOLOR= C1
1ØØ HPLOT Ø,Y TO SZ,Y
11Ø HCOLOR= C2
12Ø HPLOT Ø,Y + 1 TO SZ,Y + 1
13Ø NEXT Y
14Ø FOR I = 1 TO DLAY: NEXT I
15Ø NEXT C2
16Ø NEXT C1
17Ø END
```

Listing 9-4. Dithering.

You should see a rectangle in the upper left corner of the screen which continually changes color, and in the process takes on some shades other than the basic six. You may change the speed by altering the value of DLAY in line 3Ø, and the size of the rectangle is determined by the value of SZ in line 4Ø.

All the program does is plot pairs of horizontal lines in two alternating colors (C1 and C2), then change one or both colors and do it again. As the routine cycles, you can see the basic colors—green, blue, violet, orange, black, and white—but also some other colors such as light green and hot pink. The extra colors are generated by two colors blurring together to form a mixture, for instance white and green blurring to form light green.

There are a great many variations used to produce other shades. For example, you might mix the colors orange and black, by alternating them in a checkerboard pattern to produce dark orange, or perhaps by drawing a pure orange line followed by a line which alternates orange and black. The possibilities are limited only by your imagination.

There are a few considerations, however. For instance, if you tried to alternate green and violet dots across the screen, as in the following short program, you would run into severe problems.

```
1Ø HGR
2Ø C1 = 2:C2 = 1
3Ø FOR X = Ø TO 278 STEP 2
4Ø HCOLOR= C1: HPLOT X,Y: REM VIOLET
5Ø GOSUB 1ØØ
6Ø HCOLOR= C2: HPLOT X + 1,Y: REM GREEN
7Ø GOSUB 1ØØ
8Ø NEXT X
9Ø END
1ØØ FOR I = 1 TO 2ØØ: NEXT I: RETURN
```

Listing 9-5.

Thanks to the delay loop in line 100, you will be able to see all the action when the program runs. The first dot appears in its proper violet, but then the next dot comes on—not as the expected green, but as white, and the first dot turns white with it. The remaining dots also display white, regardless of the color set in the program. Remember, any time two consecutive dots in a row are on, they display white, so if you wish to alternate colors across a line, you must leave one or more black dots between each color and the next (see Figure 9-5).

Figure 9-5. Alternating colors.

Even if you leave black dots between colored dots, you are still not completely safe. The next listing ostensibly plots violet in column 0, blue in column 2, violet in column 4, blue again in column 6, and so on across the line.

```
10 HGR
20 C1 = 2:C2 = 6
30 FOR X = 0 TO 276 STEP 4
40 HCOLOR= C1: HPLOT X,Y: REM VIOLET
50 GOSUB 100
60 HCOLOR= C2: HPLOT X + 2,Y: REM BLUE
70 GOSUB 100
80 NEXT X
90 END
100 FOR I = 1 TO 200: NEXT I: RETURN
```

Listing 9-6.

When the program runs, a violet (group 1) dot will appear in column 0, then a blue dot will appear in column 2. Since blue is in group 2, the entire byte will become group 2 and the violet dot will change to blue also. Next, a violet dot will be plotted in column 4, turning the previous two dots violet, and then a blue dot will be plotted in column 6 to turn everything blue again. Column 8 is in the next byte of memory so when the violet dot is plotted there, the previous four dots are unaffected. Column 10 displays blue, and then column 12 displays violet so that the three dots in that byte are all violet. Column 14 begins the third byte, and, as it displays blue, the previous three dots remain violet. The final pattern is a line which has alternating violet and blue segments.

This demonstrates a propensity of Hi-Res graphics called "clashing." **Clashing** is the inadvertent switching of a dot from one color group to the other and occurs when you attempt to plot colors from opposite color groups within the same byte—the computer simply cannot and will not do it—so the attempt turns the first colors plotted into colors from the second group plotted. You cannot mix green with orange or blue in the same byte, nor can you put orange or blue in the same byte as violet. Clashing is

one of the facts of Hi-Res life, and it is virtually impossible to avoid completely, though careful planning of your figures will help you to minimize it.

Wrapping It Up

By their very nature, the topics discussed in this chapter are perhaps the most confusing in the book. The color bit controls the colors available to the rest of the byte, and the color of an individual dot is determined by whether it is in an odd or even column of the screen, and by the setting of the color bit. To display a given color, you must first determine whether it is in color group 1 or 2, and then whether it is available on an even dot or an odd dot. There is no way to display two different color groups in the same byte at all, and no way to display two colored dots next to each other without both dots turning to white.

We have seen that, despite what the manuals say, there are 560 dots across the Hi-Res screen, though you may display a maximum of 280 dots at any time. If you are working in color, you are further limited to 140 positions since each color is available on only odd or only even dots. When you are digitizing the data for a figure, you are forced to think in terms of having only 40 bytes across each line. The color groups and the odd/even availability of each color is summarized in Figure 9-6.

	Odd	Even
Group 1:	GREEN	VIOLET
Group 2:	ORANGE	BLUE

Figure 9-6.

Vocabulary

Clashing

Color Bit

Color Group

Dithering

Group 1 Colors

Group 2 Colors

Exercises

1. Use the POKE statement to display the following colors at the given location on the screen. It is assumed that column Ø is the left-most column of dots.

   a. Violet in the tenth row and twentieth column of dots.

   b. Orange in row ten, column 71.

   c. Blue in row ninety, column 1Ø4.

   d. Green in row 16Ø, column 279.

2. Write a BASIC program to cycle through each of the eight values for HCOLOR and plot a line of that color, followed by a line which alternates that color with black, and creates a rectangle containing 15 pairs of such lines. The routine needs to account for the fact that green and orange may only be plotted on odd columns, and violet and blue only on even.

3. Take the butterfly designed in Exercise 3 of Chapter 8 and display it in color.

10

Shaping Up

Objectives

After reading Chapter 10 you should be able to (if forced):

- Draw a vector diagram for a figure.
- List and encode the vectors from the diagram.
- Group the vector codes into bytes.
- Develop a shape table from one or more shapes.
- Save and recover a shape table from disk or tape.
- Use the shape commands to plot, scale, and rotate a shape.

In this chapter you will investigate the properties of Apple shapes and shape tables. The term "shape" is used here in a very specific sense. From the programmer's point of view, a shape could be almost any figure—a spaceship, frog, alien, and so on—but to the computer a shape is a set of highly formatted data which lets the machine produce a display on the screen.

Though using a shape can cost you extra time at the outset, in practice it makes drawing and animating figures much faster and simpler than the techniques we have discussed in the previous two chapters. Shapes may also be used to simulate text on the Hi-Res screen.

After you digitize a figure to create a shape, that shape can be plotted on the screen as a unit so that you no longer need to concern yourself with calculating the necessary addresses, nor the values to POKE. APPLESOFT takes care of all those grubby little details, and also plots the shape faster than you could if you used the READ and POKE technique discussed in earlier chapters.

We will show you how to digitize a figure into a shape, put the shape into a shape table, and how to save the results of your labor. From there, you will use APPLESOFT to draw, erase, rotate, and scale your shape. You will see very quickly how tedious and time-consuming shapes can be if you do them yourself. Fortunately, for your sanity, any good Hi-Res graphics editor will contain a provision for making shapes from the figures you develop on the screen, so that the machine does all the menial calculations instead of you. We highly recommend using a Hi-Res editor to make shapes.

To the Vector Belong the Spoils

To draw a shape, the Apple II uses a series of vectors which tell it whether or not to plot the current point, and which direction to move to find the next point. Each vector is visually represented by a little arrow with a dot on one end. (See Figure 10-1B.) We will begin with a very simple figure in order to illustrate the technique and rules for digitizing a shape. Figure 10-1A is the block diagram of a figure which uses only three dots, and Figure 10-1B shows the six vectors used to turn them on.

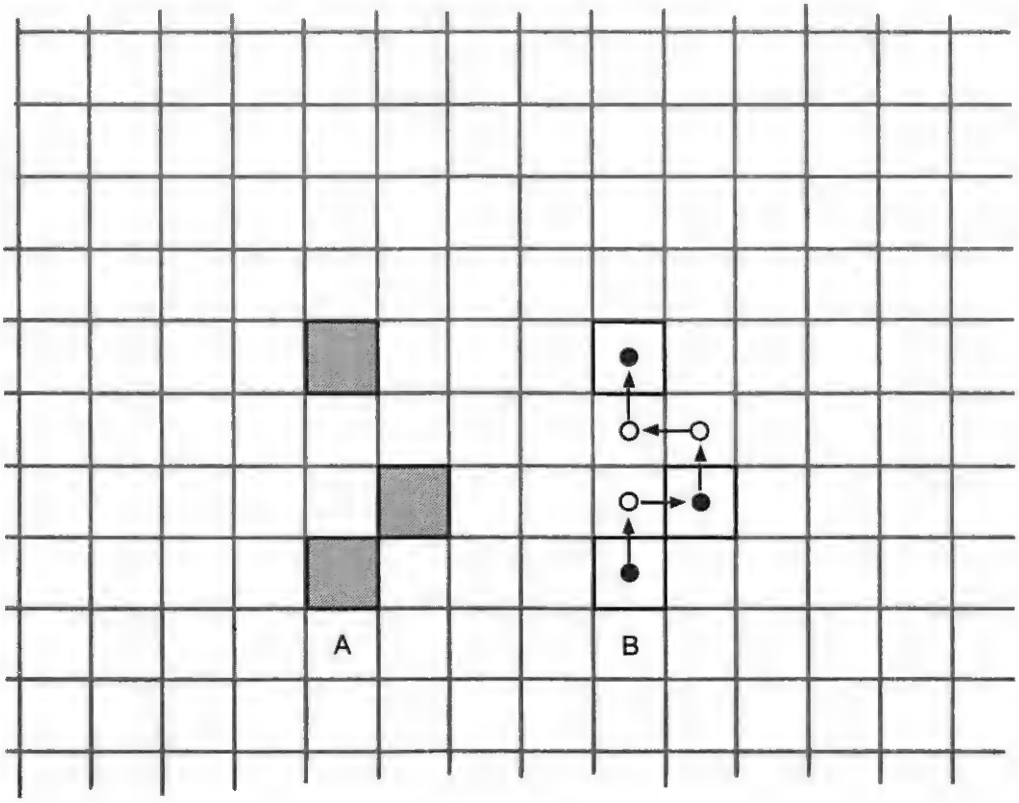


Figure 10-1.

A closed dot at the beginning of a vector indicates that the point is to be plotted, and an open dot indicates that the point is to be left alone (skipped), and the arrow points the direction to the next point. It is vital to remember that the plotting or skipping comes before the moving. We will use "P" to represent a dot which is to be plotted, and "S" to represent one which is skipped. Also, "U" means that we move up, "R" right, "D" down, and "L" left. It requires both a plot/skip and a move command to represent each vector. For example, SD represents a vector which skips the current point and moves down. With that in mind let's attempt to make sense out of Figure 10–1B.

Starting at the beginning, the bottom block, we will trace the path of the vectors to its terminus. The first vector indicates that the point is to be plotted, and then we are to move up (PU). The second vector leaves the dot alone and moves right (SR). The third vector plots and moves up (PU), while the fourth, fifth, and sixth skip and move left (SL), skip and move up (SU), and plot and move right (PR), respectively. You end up with this list of vectors:

PU
SR
PU
SL
SU
PR

The last vector plots before it moves, and the direction of movement is unimportant. Right was chosen arbitrarily—the important characteristic of that vector was the plot to turn on the last point.

The computer generates the shape on the screen by following this series of instructions: plot-up, skip-right, plot-up, skip-left, skip-up, and plot-right. You need to encode those vectors for the machine using binary codes for each instruction as shown in Figure 10–2.

```
Plot : 1          Up : ØØ
Skip : Ø       Right : Ø1
                Down : 1Ø
                Left : 11
```

Figure 10–2. Vector codes.

To encode the first vector (PU), you would use 1 for Plot and ØØ for Up to get 1ØØ. The second vector (SR) yields ØØ1, and the third (PU) gives you 1ØØ again. Take a piece of paper and write down the six vectors and their related codes. You should end up with:

```
PU : 1ØØ
SR : ØØ1
```

```
PU : 1ØØ
SL : Ø11
SU : ØØØ
PR : 1Ø1
```

To communicate those values to the computer, you need to put them together to form "byte size" pieces. For the purpose of encoding the vectors, each byte is divided into three sections, A, B, and C, as shown in Figure 1Ø–3.

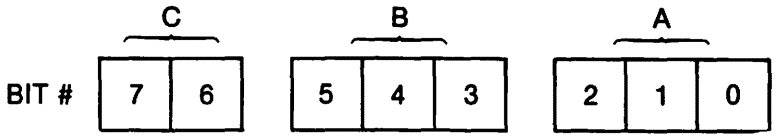


Figure 1Ø–3. Byte divisions.

Begin by placing the first vector code in section A, the second in B, and the third (if possible) in C. It is NOT permissible to break a code in the middle and carry the remainder over to the next byte; you have no doubt noticed that section C contains only two bits, so it cannot contain a complete code. The logical conclusion is that section C is unusable, and you may always leave it blank (ØØ). However, in certain cases section C can be used; the left hand bit of any code in section C is assumed to be Ø, and only the two bits on the right (the direction bits) are entered. For example, 1Ø ØØØ 1Ø1 would cause the computer to execute PR (1Ø1), SU (ØØØ), and then SD (1Ø).

Any vector placed in section C must be a "skip" vector; the third vector in our shape is a plot vector so it will not fit in section C. Therefore, it must be placed in section A of the next byte. When plotting the shape, the computer will ignore the two zeros at the left end of a byte. In fact, any zero section of a byte is ignored if all of the following sections of that byte are zero also. For example, with ØØ ØØØ Ø11, the computer will skip and move left (Ø11), then ignore the rest of the byte, and begin the following byte. It follows, then, that section C can never execute an up vector (ØØ), since it would be ignored as in ØØ 111 ØØØ where the computer would skip-up (ØØØ), plot left (111), and then ignore section C. In that case, the skip-up vector would have to be placed in section A of the next byte.

Refer to Figure 1Ø–4 as we group the vectors for our shape.

	C	B	A	C	B	A	HEX
BYTE # 1	—	SR	PU	00	001	100	0C
2	—	SL	PU	00	011	100	1C
3	—	PL	SU	00	111	011	3B
4	—	—	—	00	000	000	00

Figure 1Ø–4.

In the first byte, section A contains the PU code, section B the code for SR, and section C is set to ØØ to be ignored. The next PU code is in section A of byte number 2, followed by the SL in section B, and ØØ in C, which is again ignored. The SU is in section A of byte number 3, followed by PR and then ØØ. Byte number four is filled with zeros and marks the end of the shape. Any shape is ended by the first zero byte encountered; therefore, if you intend to put the skip-up code (ØØØ) in sections A and B, section C MUST NOT BE ZEROS!

Following these rules, Figure 1Ø–4 shows the entire shape encoded in four bytes. The shape may not be displayed until it is entered in a shape table, but before we do that, let's digitize one more shape for practice.

UFO Time

If you did not believe in flying saucers before, you will soon, as you are about to create one. Figure 1Ø–5A contains the block diagram of the saucer you will create and Figure 1Ø–5B contains its vector diagram.

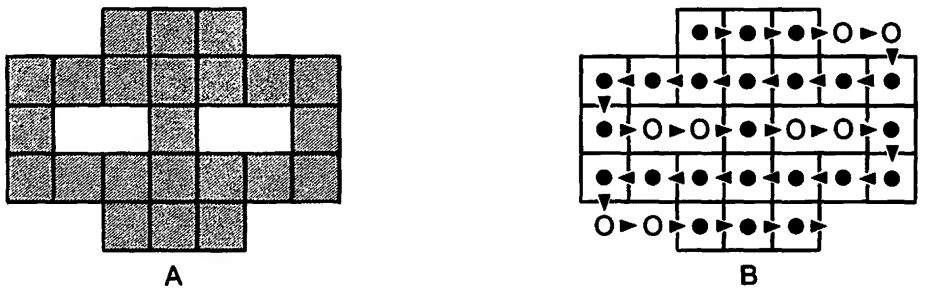


Figure 1Ø–5. Saucer.

It is necessary to plot each of the 23 dots shown in Figure 1Ø–5A, but the order in which they are traced is not important. However, changing the pattern of the vectors will also alter the values used in the shape, so for the sake of consistency we suggest that you use the vectors shown in Figure 1Ø–5B. Follow the vectors and write down a list of their functions as you go. Your list will look like this:

```
PR, PR, PR, SR, SD, PL, PL, PL, PL,
PL, PL, PD, PR, SR, SR, PR, SR, SR,
PD, PL, PL, PL, PL, PL, PL, PD, SR,
SR, PR, PR, PR
```

Go back and double check your list; accuracy is tedious but imperative!!

The next step is to encode the instructions as bytes. We have done this in Figure 1Ø–6. To be on the safe side, the encoding is done in a series of small steps, each of which is easily checked. The extra care taken now will

help minimize errors which can cost you hours of time and frustration, and will reduce the urge to physically abuse your computer.

C B A	C B A	HEX	DECIMAL
-- PR PR	ØØ 1Ø1 1Ø1	2D	45
SD SR PR	1Ø ØØ1 1Ø1	8D	141
-- PL PL	ØØ 111 111	3F	63
-- PL PL	ØØ 111 111	3F	63
-- PL PL	ØØ 111 111	3F	63
SR PR PD	Ø1 1Ø1 11Ø	6E	11Ø
SR PR SR	Ø1 1Ø1 ØØ1	69	1Ø5
-- PD SR	ØØ 11Ø ØØ1	31	49
-- PL PL	ØØ 111 111	3F	63
-- PL PL	ØØ 111 111	3F	63
-- PL PL	ØØ 111 111	3F	63
SR SR PD	Ø1 ØØ1 11Ø	4E	78
-- PR PR	ØØ 1Ø1 1Ø1	2D	45
-- -- PR	ØØ ØØØ 1Ø1	Ø5	5
-- -- --	ØØ ØØØ ØØØ	ØØ	Ø

Figure 10-6.

First, the instructions are inserted into the bytes by sections, and then each instruction is converted to its binary code. After that, the resulting byte is converted to hex and decimal. You now have digitized the saucer, but as with the first shape, you must insert it in a shape table in order to display it.

Setting the Table

A **shape table** is a collection of the digitized data for one or more shapes, up to a maximum of 255, plus some extra information at the beginning which is needed by the machine. The first thing your Apple needs to know is the number of shapes contained in the table, and the next thing that it will ask is where each of the alleged shapes is located within the table. We will use Monitor to set up a shape table containing our two shapes at memory location $3ØØ. From Monitor begin with

```
3ØØ: Ø2 ØØ Ø6 ØØ ØA ØØ ØC 1C
```

The 3ØØ: tells Monitor to begin filling memory at address $3ØØ. The first byte (Ø2), gives the number of shapes in the table, while the second byte

(ØØ) has no meaning, but it must be there to hold the place. The next two bytes (Ø6 ØØ) are used by the system to locate the first shape in the table, and the pair following (ØA ØØ) locate the second shape. (Remember that your Apple eats addresses in lo-byte/hi-byte form.) The next byte (ØC) is the beginning of the first shape. Each shape in a table is located by a pair of bytes, and the first shape begins directly following the last pair of locating bytes.

Continue to enter the shape table by typing:

```
3Ø8: 3Ø ØØ 2D 8D 3F 3F 3F 6E
     69 31 3F 3F 3F 4E 2D Ø5 ØØ
```

The second byte of this line (ØØ) is the end of your first shape, and the second shape follows immediately, beginning with 2D and ending with another ØØ.

Finding Your Shapes

If you put your finger at the beginning of the table (value Ø2) and then count in, you will find that the first shape begins with the sixth byte of the table as we promised when we entered Ø6 ØØ for the first pair of location bytes. (The Ø2 counts as the zeroth byte of the table.) The location bytes give the location of the corresponding shape RELATIVE TO THE BEGINNING OF THE TABLE. A value used to specify a location relative to some fixed address is called an **offset.** The second pair of location bytes is ØA ØØ, and since $A equals 1Ø that claims that the second shape begins with the tenth byte of the table. When you key in a table on your own, you may initially put zeros in the location bytes for each shape, and after the rest of the table is entered and the locations are fixed, go back to the location bytes and enter the correct offsets.

Shapes tend to be long (and drawn out?), and it is not unusual to have a large number as an offset, such as 2Ø85. To enter that value in the location bytes, first turn it into a hex number, 2Ø85 equals $Ø825. Divide this number into lo-byte ($25) and hi-byte ($Ø8). The first byte for the offset would be $25, and the second $Ø8. (Remember lo-byte/hi-byte.)

Using Draw

If you have worked with your own shapes, at this point you would have spent a great many hours in designing, digitizing, and entering the data (a marvelous form of self punishment!). You are finally ready to do one of three things: (1), display the shapes; (2), forget the whole thing or (3), assign the project to someone you detest! We will select option one.

If you have been following along, you presently have the shape table entered at $3ØØ. Prior to drawing the shapes, you need to inform your Apple as to the table's whereabouts by placing the starting address of the table at memory locations 232 and 233 ($E8 and $E9). APPLESOFT always

looks in those locations to find the starting address of the current shape table, and if you forget to load the proper address, there is no telling what is going to pop up on the display. Since you are still in Monitor, type

```
E8: ØØ Ø3
```

This will put the address $Ø3ØØ (remember lo-byte/hi-byte) in the proper location. Finally, key in and run the following program.

```
1Ø REM SHAPE DEMO
2Ø REM
3Ø ROT = Ø: REM SET ROTATION TO Ø
4Ø S = 1
5Ø HGR: HCOLOR = 3
6Ø FOR I = 7Ø TO 19Ø STEP 3Ø
7Ø SCALE = S: REM SET SCALE TO S
8Ø DRAW 1 AT I,3Ø
9Ø DRAW 2 AT I,1ØØ
1ØØ S = S + 1: REM INCREMENT S
11Ø NEXT I
12Ø END
```

Listing 10-1.

Line 3Ø sets the rotation of the figure to Ø (no rotation) so it will be drawn as it was designed, and line 5Ø sets the Hi-Res graphics mode and then sets HCOLOR to 3 (white1). Line 7Ø sets the scaling factor to the present value of S; the scaling factor will increment from one through five. At SCALE = 1, the figure is the same size as it was designed, and at SCALE = 5, it is five times the original. The SCALE may be set to any number from zero to 255, but, as you will see, increasing the SCALE tends to greatly distort the shape. If you wish, you may set SCALE = Ø, but zero corresponds to 256, and the shape will be enlarged beyond recognition.

Line 8Ø DRAWs shape number 1 beginning at the coordinates given by I,3Ø and line 9Ø DRAWs shape 2 at I,1ØØ. The coordinates denote the placement of the point of origin for the shape—the place where the chain of vectors began.

If everything was entered as specified, the program will draw five copies of both shapes, and each copy will increase in size.

Saving Shape Tables

If you have a disk drive, you may save the table by using BSAVE to save the range of memory which contains it. Remember to save the final ØØ with the rest of the table, otherwise you will get some unpredictable results

as the computer draws off the end of your shape and on into memory garbage. To save the table currently in memory, type:

```
BSAVE TABLE1,A$300,L$19
```

or

```
BSAVE TABLE1,A768,L25
```

The only difference between the two is that the first gives the address (A) and length (L) in hex, and the second gives them in decimal. If you add the following lines to Listing 10-1, they will automatically load the table and inform the Apple of its location when the program is run.

```
22 PRINT CHR$(4);"BLOAD TABLE1"
24 REM CHR$(4) IS CTRL-D
26 POKE 232,0: POKE 233,3
28 REM $0 INTO $E8, $3 INTO $E9
```

A longer shape table may not fit at $300 since only the range $300-$3BF is available; a total of 192 bytes. (See the memory map in Chapter 3.) Long tables are often placed up at the top of available memory, and then HIMEM set underneath them so they will not be overwritten by the variables of your APPLESOFT program. For a 48K machine, DOS sets HIMEM to $9600 (38400), so if your shape table was $100 (256) bytes long you would

```
BLOAD [tablename],A$9500
HIMEM: 38144
```

or, in decimal,

```
BLOAD [tablename],A38144
HIMEM: 38144
```

A more thorough discussion of the use of BLOAD and HIMEM can be found in Chapter 7 where we covered saving the Low-Res screen.

If you wish to save the table on tape, you must enter the Monitor and use a variation of the Write command to save the memory range containing the table. You must know the hex values for the first address, the last address, and the length of the table. In the table that contains the saucer, this would be $300, $318, and $19, respectively. First, you put the length of the table in locations 0 and 1 by typing:

```
0: 19 00 (Remember lo-byte/hi-byte)
```

then write the information onto the tape using the Monitor "W". Write bytes zero and one onto the tape (they contain the length of the table), and then immediately write the table itself. To do all this in one instruction, you could type:

```
0.1W 300.318W
```

Remember to let the tape run for a moment on the record setting between typing the second "W" and pressing the return key.

You may use the Monitor READ command to reload the table, but APPLESOFT has a much better method using the SHLOAD command. SHLOAD loads the shape table from tape and stores it just below HIMEM. HIMEM is set below the table to protect it, and the starting address of the table is placed in $E8 and $E9. The beauty of this process is that all the details are taken care of automatically. The only command you give is SHLOAD, and, unlike the Monitor command, SHLOAD may be used from within a BASIC program. For our example, rewind your shape table tape, push the play button, type

```
SHLOAD
```

and everything is done for you. TA DA!!

The table may also be loaded directly as part of the program by using the POKE statement. Listing 1Ø-2 POKEs the data for our shape table, asks you to specify the scaling factor (1-255), and then draws the saucer on the screen with ten different rotation values.

Learning by ROT

ROT=Ø causes the shape to be oriented in the same way as it was defined—no rotation. When ROT=16, DRAW will show the shape rotated 9Ø degrees clockwise from the original, ROT=32 rotates 18Ø degrees, and ROT=48 rotates it 27Ø degrees clockwise (9Ø degrees counter-clockwise). ROT=64 is back where it started, and so forth in increments of 16 all the way through 256.

The ROTation parameter is partially dependent on the setting of SCALE. According to the APPLESOFT manual, at SCALE=1, only the four rotation values [Ø,16,32,48] are recognized. However, as you can see by using Listing 1Ø-2, the values [8,24,4Ø,56] also work at SCALE=1. At SCALE=2, sixteen rotations are available, with [4,12,2Ø,28,36,44,52,6Ø] being the additional values. The larger the scale, the more rotations available, up to a point. If you set ROT to a value it does not recognize, it will rotate the shape to a value it does recognize.

Hopefully, this program will demonstrate and clarify all of those rules concerning ROT. Please run the program and watch the values of ROT at the bottom of the screen, and the effects they have on the figure. You will have to press reset to end the program.

```
1Ø REM DEMONSTRATE
2Ø REM POKING TABLE
3Ø REM
4Ø REM POKE ADDRESS
5Ø REM OF TABLE
```

```
60 POKE 232,0: POKE 233,3
70 REM
80 REM READ AND POKE
90 FOR L = 768 TO 789
100 READ V: POKE L,V
110 NEXT L
120 REM
130 REM INITIALIZE
140 REM
150 HOME : VTAB 21
160 HGR
170 HCOLOR= 3
180 INPUT "SCALE? ";S$: SCALE=
    VAL (S$)
190 R = 0
200 REM
210 REM BEGIN LOOP
220 REM TO PLOT SHAPES
230 REM
240 FOR I = 10 TO 244 STEP 26
250 ROT= R
260 DRAW 2 AT I,100
270 REM PRINT ROT VALUES
280 PRINT " ";R;" ";
290 R = R + 2: REM INCREMENT ROT VALUE
300 NEXT I
310 PRINT
320 PRINT "ROTATION VALUES"
330 INPUT "PRESS RETURN TO
    CONTINUE";R$: GOTO 150
340 END
350 REM
360 REM DATA TABLE
370 REM
380 DATA 2,0,6,0,10,0,12,28,48,0,45,
    141,63,63,63,110,105,49,63,63,63,
    78,45,5,0
```

Listing 10-2.

An Apple a Day...

The following section creates a shape from our perennial apple, and demonstrates how to place the point of origin anyplace you like in a shape. Only the first several vectors are digitized, then Listing 10-3 gives you the entire shape as a shape table in order to spare you the trouble. For (relative) simplicity, the shape is drawn by plotting every dot instead of every

second dot, so you would expect to always have a white apple (remember that consecutive dots display white!), but you will soon see differently. Figure 10-7A gives the block diagram for the shape, and 10-7B shows the beginning of the vector diagram.

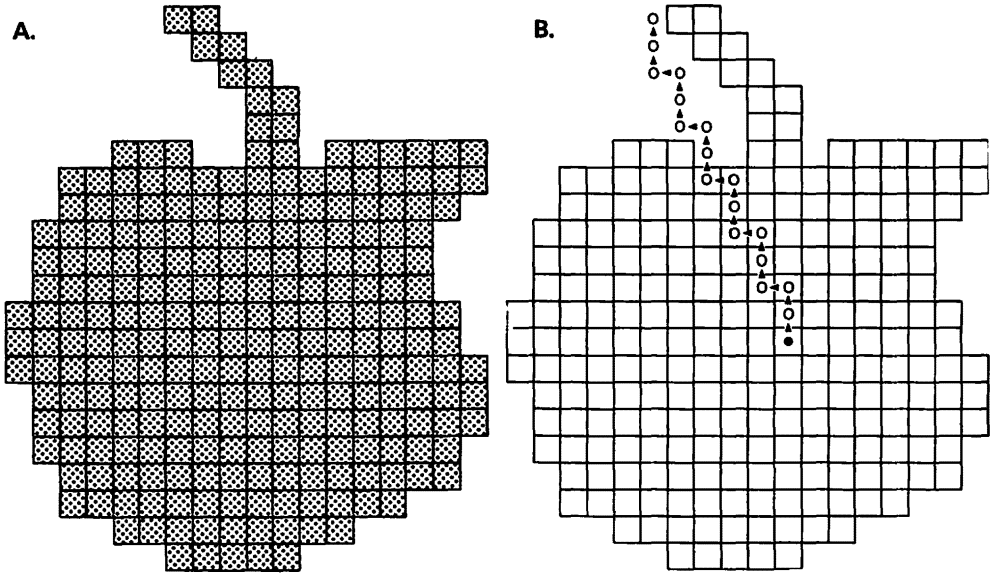


Figure 10-7. Apple figures.

The vectors plot one point within the apple, and then, without plotting, move to the top of the stem. A figure is always ROTated around its origin, and by placing the point of origin as we have, the apple will rotate about its center instead of the top. The only reason to plot the origin is to show its location, and it could just as well have been skipped. The first twenty vectors are:

```
PU SU SL SU SU SL SU SU SL SU
SU SL SU SU SL SU SU SR PR PD
```

Figure 10-8 gives you the first portion of the digitization.

C B A	C B A	HEX	DECIMAL
SL SU PU	11 000 100	C4	196
SL SU SU	11 000 000	C0	192
SL SU SU	11 000 000	C0	192
SL SU SU	11 000 000	C0	192
SL SU SU	11 000 000	C0	192
SR SU SU	01 000 000	40	64
-- PD PR	00 110 101	35	53
-- PD PR	00 110 101	35	53
. . .	. . .	.	.
. . .	. . .	.	.
. . .	. . .	.	.

Figure 10-8.

Sections A and B contain the code to move up (ØØ) for the first six bytes, and section C contains the code to move left (11) for the first five, while the sixth causes a move to the right. It may seem pretty silly to move to the left in number 5 and then immediately right in number 6 instead of moving straight up in both of them. Remember, however, that if number 5 were only to move up, it would be all zeros—ØØ ØØØ ØØØ—and would erroneously signal the end of the shape, so the small deception is necessary. The remainder of the shape is digitized in the same manner as the saucer, so instead of belaboring the point, just enter Monitor and type the shape table given in Listing 1Ø-3.

```
3ØØ: Ø1 ØØ Ø4 ØØ C4 CØ CØ CØ
3Ø8: CØ 4Ø 35 35 35 35 2B 4D
31Ø: Ø9 89 3F 3F FF FF 3B FF
318: 2A 2D 2D 2D 2D 2D 2D 2D
32Ø: 1E 3F 3F 3F 3F 3F 3F 3F
328: 17 2D 2D 2D 2D 2D 2D 2D
33Ø: 3E 3F 3F 3F 3F 3F 3F 37
338: 2D 2D 2D 2D 2D 2D 2D 15
34Ø: 3F 3F 3F 3F 3F 3F 3F 3F
348: 2E 2D 2D 2D 2D 2D 2D 2D
35Ø: 2D 15 3F 3F 3F 3F 3F 3F
358: 3F 3F 3F ØE 2D 2D 2D 2D
36Ø: 2D 2D 2D 2D 3E 3F 3F 3F
368: 3F 3F 3F 3F 37 2D 2D 2D
37Ø: 2D 2D 2D 2D 35 3F 3F 3F
378: 3F 3F 3F 3F 2E 2D 2D 2D
38Ø: 2D 2D F5 3B 3F 3F 3F 4E
388: 2D 2D Ø5 ØØ
```

Listing 1Ø-3. Apple shape table.

The first byte declares that there is one shape in the table, and the second byte is unused. The third and fourth bytes give you the offset for the first (and only) shape: 4 (remember lo-byte/hi-byte), and the following byte is the beginning of the shape.

Save the table (if you have a disk) by typing

```
BSAVE APPLESHAPE,A$3ØØ,L$8C
```

To see what you have accomplished, type

```
POKE 232,Ø:POKE 233,3
HGR:SCALE = 1:ROT =Ø
DRAW 1 AT 1ØØ,1ØØ
```

The first line tells the system where to find the table, the second is obvious, and the third draws the shape from that table on the Hi-Res screen. If you managed to get all of the numbers keyed in correctly, you will see a little

white apple in the center of your screen. If this is not so, enter Monitor and list the memory range for the table by typing

```
300.38F
```

and check your numbers against the ones in Listing 10-3.

If the apple did turn out correctly, let's label it using a second shape table. From within Monitor, place the new table at memory location $4000 by typing

```
4000: 04 00 0A 00 21 00 2E 00
4008: 36 00 24 24 24 24 24 0C
4010: 0C 2D 6D 15 36 36 3E 3F
4018: 3F 06 49 09 36 36 36 36
4020: 00 24 24 2C 2D 35 36 36
4028: 3F 3F 36 36 36 00 24 24
4030: 24 24 24 24 24 00 24 24
4038: 2C 2D 35 3E 3F 96 2D 2D
4040: 00

3D0G
```

Listing 10-4. Apple text (courtesy of Dean Lewis).

This table contains four shapes representing the letters A, P, L, and E. Save this table (again, only if you have a disk) by typing

```
BSAVE APPLE TEXT,A$4000,L$41
```

We can use APPLE TEXT to simulate putting a little text on the Hi-Res screen. There is a continuing debate about whether you can put true text on a graphics screen, other than that in the text window at the bottom. The Apple manuals say absolutely not, and they ought to know—it's their machine. However, Bob Bishop says "yes," and he ought to know—he's done it. The technique is very complex, difficult to put into practice, and impossible for the BASIC programmer, so it will not be covered in this text. We refer interested readers to the October 1982 issue of *SOFTALK* magazine, pages 54 through 63.

The BASIC programmer can do almost as well by designing shapes to represent any or all of the letters, as we have done with APPLE TEXT. So let's put text on the Hi-Res screen by typing

```
BLOAD APPLESHAPE
BLOAD APPLE TEXT
HGR:SCALE = 1: ROT = 0
POKE 232,0:POKE 233,3
DRAW 1 AT 100,100
POKE 232,0:POKE 233,64
DRAW 1 AT 115,120
```

```
DRAW 2 AT 130,120
DRAW 2 AT 140,120
DRAW 3 AT 150,120
DRAW 4 AT 156,120
```

The fourth line puts the address of the apple shape table in locations 232 and 233, then the fifth line DRAWs from that table. Line number 6 then places the address of the APPLE TEXT table in the proper locations so that the remaining commands DRAW from it. Location $4000 is $00 and $40 in lo-byte/hi-byte form, and $40 = 64.

You can see how simple it is to put "text" on the Hi-Res screen, and also to DRAW from two or more shape tables which are stored in memory at the same time.

Since things seem to be going so smoothly, let's make them a bit more complex. Key in and run the program in Listing 10-5. To save time, we suggest that you dispense with the 29 REM statements.

```
10 REM BACKGROUND DEMO
20 REM
30 REM
40 REM INITIALIZE
50 PRINT CHR$(4);"BLOAD APPLESHAPE"
60 SCALE= 1
70 POKE 232,0: POKE 233,3
80 FOR I = 0 TO 7: READ CLR$(I): NEXT I
90 VTAB 24
100 REM
110 REM
120 REM BEGIN LOOPS TO
130 REM VARY BACKGROUND
140 REM AND SHAPE COLORS
150 REM
160 FOR C1 = 1 TO 7
170 FOR C2 = 0 TO 6
180 IF C1 = C2 GOTO 210
190 PRINT CLR$(C2);" OVER ";CLR$(C1)
200 GOSUB 280
210 NEXT C2,C1
220 END
230 REM
240 DATA BLACK1,GREEN,VIOLET,WHITE1,
    BLACK2,ORANGE,BLUE,WHITE2
250 REM
260 REM DRAWING ROUTINE
270 REM
280 HGR
290 FLAG = 0: REM FIRST PASS
```

```
300 REM
310 REM DRAW BACKGROUND
320 HCOLOR= C1
330 FOR Y = 10 TO 160
340 HPLOT 10,Y TO 180,Y
350 NEXT Y
360 REM
370 REM DRAW SHAPE
380 REM
390 HCOLOR= C2
400 R = 0: REM ROTATION VALUE
410 REM
420 REM BEGIN LOOPS TO DRAW
430 REM REPETITIONS OF SHAPE
440 REM
450 FOR Y = 20 TO 100 STEP 80
460 FOR X = 20 TO 220 STEP 50
470 ROT= R: REM SET NEW ROTATION
480 DRAW 1 AT X,Y
490 R = R + 2
500 FOR I = 1 TO 200: NEXT I: REM DELAY
510 NEXT X,Y
520 REM
530 REM
540 REM FLAG = 0 IF FIRST PASS
550 REM FLAG = 1 IF SECOND PASS
560 REM IF FIRST PASS, PLOT
570 REM AGAIN IN BLACK1
580 IF NOT FLAG THEN FLAG = 1:
    HCOLOR= 0: GOTO 400
590 RETURN
```

Listing 10-5. Pandora's apple.

The program is titled "Pandora's Apple" because it displays a whole herd of unexpected and sometimes unsettling results. The function of the program is to draw the apple shape in various colors and rotations (0, 2, 4, ..., 18) superimposed on a background of a different color, and then "erase" the shape by drawing over it in black1. The background does not completely fill the screen so that you may also see the shape plotted on the black1 background. The techniques used in the program are standard and we will not mess with them since the results are the important topic.

How Do You Like Them Apples?

The program begins by painting a green field, and then DRAWing the shape over it in black1—nothing unusual so far. The colors used are

printed at the bottom of the screen so you can keep up with the action. The next pass DRAWs violet on green, and around the border of the apple you should notice white spots (mold?) caused by the violet dots of the apple plotting next to the green dots of the background. Remember, anytime you plot two adjacent dots, they display as white.

When black2 is plotted over the green, you see the effect of clashing in all its radiant glory. Any time that a group 2 dot is plotted within a group 1 byte, the entire byte becomes group 2. Hence, some of the green changes to its group 2 counterpart, orange. As the program progresses you will see several more examples, both of white edges and of clashing (Oh goody!).

When the program stops, change line 60 to read:

```
60 SCALE = 2
```

Run the program again and watch the fun! When it plots black1 over the green, it starts by only turning every other line black, and when it gets to the lower left corner of the green field, it only plots a few dots! The reason for this, and later shennanigans, is that at SCALE = 2, the apple is being plotted on every second line only. By coincidence, it happens to be plotting on only the odd columns, and green does not exist on those lines.

At SCALE = 1, the shape data causes the computer to plot across one row of the apple, and then execute a plot-down to drop to the following row. At SCALE = 2, the machine executes every vector twice, including the PD vector, so it drops two lines between plotting each row.

When violet apples are plotted over the green field, the ones rotated 90 degrees (in the lower right corner of the green field) plot as white (A farmer's nightmare!). The apple is plotting on the odd columns causing the violet in those columns to be turned on while the field is made up of green in the even columns. Since you then have both the odd and even columns on, the result is white! For a similar reason, plotting rotated white1 apples on a black background produces only violet apples since turning on only the odd columns produces violet but no green.

When the program tries to DRAW rotated green apples on a group 1 background, nothing is plotted, since the position of the apple causes it to attempt to plot green on odd columns, and green exists only on even columns. Plotting green apples on a group 2 field will cause portions of that field to change to the corresponding group 1 color—another instance of clashing. The program will run through all the colors, and similar strange combinations will occur. For fun, you might try setting the scale to three and watching what that does to the innocent apple.

XDRAW

You saw in the last program that using DRAW to plot the shape over the background in one color, and erasing it by using black, always left the

colored background mangled. You could erase it using the color of the field, but that would mess up the black section of the background. The XDRAW command eliminates that problem. With Listing 1Ø-5 in memory, type

```
17Ø
18Ø
19Ø PRINT CLR$(C1)
21Ø NEXT C1
39Ø
48Ø XDRAW 1 AT X,Y
```

Run the modified program and notice the differences. Line 48Ø changes the DRAW to XDRAW, and the other statements eliminate C2 which represents the color in which the apple was DRAWn. Where DRAW attempts to plot the apple in the specified color, XDRAW ignores HCOLOR and plots the shapes in colors which depend entirely on the background. XDRAW automatically plots the shape in the complement of the background color. Black and white are complements, as are blue and green, orange and violet. XDRAWing on the green field will give you a blue apple, and any portion which overlaps into the black will be white. The useful aspect of XDRAW is that when it XDRAWs the figure the second time, it erases cleanly and completely, regardless of the background. This attribute is very handy when animating—a subject we will cover in later chapters.

Recapping

We have shown you how to design and digitize shapes, and at the same time we have hopefully shown you the value of a good Hi-Res editor. If you are planning to do any serious work in Hi-Res, an editor is a necessary tool. We have also demonstrated how to use BSAVE or the Monitor "W" to save the table, and BLOAD or SHLOAD to recover it. You have also been introduced to the commands used to plot, rotate, and scale the shape. The initial work involved with shapes is wearisome and frustrating, but when the shape is done it is a simple yet powerful element of your programming arsenal.

Vocabulary

Digitize
DRAW
Offset
Point of Origin
ROT
SCALE
Shape
Shape Table
SHLOAD
Vector
Vector Diagram
XDRAW

Exercises

1. Write a BASIC program to lower HIMEM and then load the shape table which contains the saucer in the space created above HIMEM.
2. Extend the program in Exercise 1 to DRAW the saucer at every even point between (6,100) and (200,100).
3. Repeat Exercise 2, but use XDRAW instead of DRAW.
4. At every point between (6,100) and (200,100) XDRAW the saucer twice in succession, and then move to the next point and XDRAW twice again, and so on.

11

Graphs and Charts

Objectives

After reading Chapter 11 you should be able to:

- Have your computer draw a bar, line, or circle graph using our programs and your data.
- Use the computer to graph your own equations.
- Label your graphs.

Processing data, or "number crunching" as it is sometimes called, has long been one of the strengths of the computer. In earlier machines, the results of the processing were all too often only pages and pages of numbers. Long columns of numbers often obscure the meaning those same figures are supposed to convey, so displaying data in graphic form is an important tool. Bar charts, line graphs, and circle graphs are the three most commonly used display methods, and we will give at least one example of each in this chapter. In three of the four examples, the data to be graphed is contained in BASIC DATA statements so that you need only replace the sample data with yours, and the program will graph your data. The other example graphs a mathematical equation over a specified range. Modifying that program will require you to change the formula and the scaling factors. We will talk more about each example when the time comes.

Belly Up to the Bar

Listing 11-1 creates a bar graph with the bars running either vertically or horizontally. Key the program in and run it. As usual, you may omit any or all of the REM statements.

```
10 REM BAR CHART DEMO
20 REM
30 REM ******************
40 REM * INITIALIZATION *
50 REM ******************
60 REM
70 REM DETERMINE VERT/HORIZ CHART
80 REM
90 TEXT : HOME
100 PRINT "PRESS 0 FOR A VERTICAL CHART"
110 PRINT "PRESS 1 FOR A HORIZONTAL CHART"
120 GET H$
130 IF H$ < > "1" AND H$ < > "0" GOTO 90
140 H% = VAL (H$)
150 REM
160 REM READ NUMB BAR, WIDTH BAR,
170 REM AND WIDTH SPACE
180 REM
190 READ NB%,WB%,WS%
200 REM
210 REM CHECK TOTAL WIDTH
220 REM
230 IF NB% * (WB% + WS%) < 40 GOTO 320
240 PRINT CHR$ (7); "CHART TOO WIDE"
250 PRINT "CONTINUE? (Y/N)": GET R$
260 IF R$ = "N" GOTO 670
270 IF R$ = "Y" GOTO 320
280 GOTO 240
290 REM
300 REM READ DATA & FIND MAXIMUM
310 REM
320 DIM BAR(NB%),LBL$(NB%)
330 FOR I = 1 TO NB%
340 READ BAR(I),LBL$(I)
350 IF MAX < BAR(I) THEN MAX = BAR(I)
360 NEXT I
370 REM
380 REM ESTABLISH SCALE
390 REM
400 SCL = 39 / MAX
410 REM
420 REM **************
430 REM * DRAW CHART *
440 REM **************
450 REM
460 GR
470 FOR I = 1 TO NB%
```

```
480 REM
490 REM SET NEW COLOR
500 REM
510 C% = INT (15 * RND (1) + 1)
520 IF C% = 5 OR CO% = C% GOTO 510
530 COLOR= C%
540 CO% = C%
550 REM
560 REM PLOT A BAR AND PRINT LABEL
570 REM
580 HTAB (XS + 1): VTAB (21)
590 PRINT LEFT$ (LBL$(I),WB%);
600 FOR X = XS TO XS + WB%-1
610 IF X > 39 THEN PRINT CHR$ (7);
    "CHART TOO WIDE": GOTO 670
620 IF H% THEN HLIN 0,BAR(I) * SCL AT X
630 IF NOT (H%) THEN VLIN 39,39-BAR(I) * SCL AT X
640 NEXT X
650 XS = X + WS%
660 NEXT I
670 END
700 REM
705 REM **************
710 REM * DATA TABLE *
715 REM **************
720 REM
730 DATA 4,5,2
740 DATA 100,'80
750 DATA 80,'81
760 DATA 50,'82
770 DATA 40,'83
```

Listing 11-1. Bar graphs.

This routine draws either horizontal or vertical bars. Select the vertical bar graph option when the computer screen displays:

```
PRESS 0 FOR A VERTICAL CHART
PRESS 1 FOR A HORIZONTAL CHART
```

You will see the bar chart appear with labels underneath to indicate years. If you run the program again and select a horizontal graph, the years still appear at the bottom. Why? This happens because it is not possible to put text labels in the body of the Low-Res screen. As you will see in Listing 11-3, labels are often unique to each graph, and many times they have to be added after the graph is done, and in a less than elegant manner.

Let's have more fun with this program before we continue. Change the third data value in line 730 from 2 to -1 and run the program another time. The item of data you changed specifies the spacing between each bar, and setting a space of -1 causes the bars to overlap! The second item in line 730 sets the width of each bar; try running the program with different values for the width.

The first item on line 730 tells the program how many bars are going to be used. You may change that value, but you must be sure to have data for each bar you specify in that entry. The value to be plotted by each bar, and the label associated with that bar, is given in the data pairs which begin at line 740. The first bar plots a value of 100, and is for the year '80.

It is time to take a closer look at some of the less obvious parts of this example.

```
230 IF NB% * (WB% + WS%) < 40 GOTO 320
240 PRINT CHR$ (7);"CHART TOO WIDE"
250 PRINT "CONTINUE? (Y/N)": GET R$
260 IF R$ = "N" GOTO 670
270 IF R$ = "Y" GOTO 320
280 GOTO 240
```

Line 230 calculates the width of the chart and warns you if it is too wide. You have the option to continue nonetheless.

```
320 DIM BAR(NB%),LBL$(NB%)
330 FOR I = 1 TO NB%
340 READ BAR(I),LBL$(I)
350  IF MAX < BAR(I) THEN MAX = BAR(I)
360 NEXT I
```

This section reads the data and label for each bar. Line 350 causes MAX to end up containing the maximum data value to be plotted. The maximum must be known in order to establish a scale for the graph.

```
400 SCL = 39 / MAX
```

Line 400 uses the maximum to establish a scaling factor, SCL. By setting SCL equal to the ratio 39/MAX, we cause the tallest bar to be 39 units tall. By making the constant smaller, say 25, you can reduce the height of the bars. You may increase the constant to 40, but any larger value will cause the program to try to plot out of its range since the Low-Res screen is only 40 by 40.

```
470 FOR I = 1 TO NB%
```

This begins the loop which executes once for each bar to be plotted.

```
510 C% = INT (15 * RND (1) + 1)
520 IF C% = 5 OR CO% = C% GOTO 510
530 COLOR= C%
540 CO% = C%
```

These lines select a random color for each bar. Line 52Ø prevents the new color (C%) from being the same as the previous color (CO%) or color number 5. Color number 5 is excluded because there are two greys in Low-Res (colors 5 and 1Ø).

```
58Ø HTAB (XS + 1): VTAB (21)
59Ø PRINT LEFT$ (LBL$(I),WB%);
```

These position the cursor at the correct place and print as much of the label as will fit in the width of the bar.

```
6ØØ FOR X = XS TO XS + WB%-1
61Ø IF X > 39 THEN PRINT CHR$ (7);"CHART TOO WIDE":
    GOTO 67Ø
62Ø IF H% THEN HLIN Ø,BAR(I) * SCL AT X
63Ø IF NOT (H%) THEN VLIN 39,39-BAR(I) * SCL AT X
64Ø NEXT X
```

This loop draws the bar. If you selected a horizontal graph, line 62Ø will execute; line 63Ø executes if you chose vertical. The length of each bar is found by multiplying the data value being graphed by the scaling factor calculated earlier. You may have wondered why the bar is drawn from 39 to 39 minus its length, instead of being drawn from zero. It must be continually taken into account that 39 is the bottom of the screen and Ø is the top, where most graphs use Ø at the bottom.

```
65Ø XS = X + WS%
```

XS represents the start of the current bar. After the bar has been drawn, this line moves the starting location to that of the next bar by adding the bar width.

A very similar routine could be written to plot the bar graph on the Hi-Res screen instead of on the Low-Res display. The limits would be changed from 39 to 159 or 279, depending on whether you plan to draw vertically or horizontally, and you would be using the HPLOT TO command instead of HLIN or VLIN.

This Is Where You Draw the Line

Sometimes data is better represented as a line graph. Listing 11-3 draws such a graph in Hi-Res and uses a shape table to put labels on the graphics screen. Enter and save the shape table by typing the following from BASIC:

```
CALL -151
3ØØ: ØB ØØ 18 ØØ 2Ø ØØ 2B ØØ
3Ø8: 36 ØØ 41 ØØ 4E ØØ 5D ØØ
31Ø: 66 ØØ 75 ØØ 81 ØØ 9Ø ØØ
318: 29 2D D8 24 24 3C 32 ØØ
32Ø: 29 2D 25 DB ØC ØC ØC 24
```

```
328: 3B 3F ØØ Ø8 15 2D ØC 24
330: 23 44 3F 3F Ø7 ØØ 49 21
338: 24 24 3C 9B 36 2E 2D ØD
340: ØØ Ø8 ØE Ø5 2D 2Ø 1C 3F
348: 1C 24 2D 2D Ø3 ØØ Ø8 24
350: 24 ØC 2D 35 92 36 3B 27
358: 18 Ø8 2D 2D ØØ 21 ØC ØC
360: ØC ØC 3C 3F 3F ØØ Ø9 2D
368: ØC 24 3B 3F 2Ø ØC 2D 15
370: 36 DB 13 36 ØØ Ø9 2D ØC
378: 24 3B 3F 2Ø ØC 2D 15 36
380: ØØ Ø1 29 2D 2Ø 24 24 3B
388: 3F 32 26 2E 28 28 28 ØØ
390: 24 24 ØØ

3DØG
BSAVE NUMBER TABLE,A$3ØØ,L$93
```

Listing 11-2. Number table.

This number table contains eleven shapes; first come the digits 1 through 9, then zero, and finally a critter resembling a hyphen which will be used to mark each axis of the graph.

Now type in and run Listing 11-3.

```
1Ø REM LINE CHART DEMO
2Ø REM
3Ø REM ******************
4Ø REM * INITIALIZATION *
5Ø REM ******************
6Ø REM
7Ø TEXT : HOME
8Ø VTAB (21)
9Ø PRINT CHR$ (4);"BLOAD NUMBER TABLE,
   A$3ØØ"
1ØØ POKE 232,Ø: POKE 233,3
11Ø SCALE= 1
12Ø REM
13Ø REM READ NUMBER OF POINTS
14Ø REM AND SPACING
15Ø REM
16Ø READ NP%,SP%
17Ø REM
18Ø REM CHECK TOTAL WIDTH
19Ø REM
2ØØ IF (NP%-1) * SP% < 28Ø GOTO 3ØØ
21Ø PRINT CHR$ (7);"CHART TOO WIDE"
```

```
220 PRINT "CONTINUE? (Y/N)": GET R$
230 IF R$ = "N" GOTO 860
240 IF R$ = "Y" GOTO 300
250 GOTO 210
260 REM
270 REM READ DATA POINTS
280 REM AND FIND MAXIMUM
290 REM
300 DIM DP(NP%-1)
310 FOR I = 0 TO NP%-1
320 READ DP(I)
330 IF MAX < DP(I) THEN MAX = DP(I)
340 NEXT I
350 REM
360 REM ESTABLISH SCALE
370 REM
380 SCL = 150 / MAX
390 REM
400 REM DRAW FRAME
410 REM
420 HGR
430 HCOLOR= 6
440 HPLOT 24,0 TO 24,159 TO 278,159 TO
    278,0 TO 24,0
450 REM
460 REM ******************
470 REM * CUSTOM SCALING *
480 REM ******************
490 REM
500 REM SCALE VERTICAL AXIS
510 REM
520 HCOLOR= 3
530 FOR I = 20 TO MAX STEP 20
540 ROT= 16: DRAW 11 AT 24,160-I * SCL
550 ROT= 0
560 N$ = STR$ (I):NL = LEN (N$)
570 FOR J = 1 TO NL
580 DIGIT = VAL ( MID$ (N$,J,1)):
    IF DIGIT = 0 THEN DIGIT = 10
590 DRAW DIGIT AT (J-1) * 7,160-I * SCL
600 NEXT J
610 NEXT I
620 REM
630 REM SCALE HORIZONTAL AXIS
640 REM
650 ROT= 0
660 FOR I = 1 TO NP%-1
```

```
67Ø X = 24 + I * SP%
68Ø IF X < 28Ø THEN DRAW 11 AT X,159
69Ø NEXT I
7ØØ VTAB (21)
71Ø PRINT " J F M A M J J A S O N D J F M
           A E A P A U U U E C O E A E A
           N B R R Y N L G P T V C N B R";
72Ø VTAB (2Ø)
73Ø REM
74Ø REM ********************
75Ø REM * PLOT DATA POINTS *
76Ø REM ********************
77Ø REM
78Ø HCOLOR= 7
79Ø HPLOT 24,16Ø-DP(Ø) * SCL
8ØØ FOR I = 1 TO NP%-1
81Ø X = 24 + I * SP%
82Ø Y = DP(I) * SCL
83Ø IF X > 279 THEN PRINT CHR$ (7);
    "CHART TOO WIDE": GOTO 86Ø: REM (END)
84Ø HPLOT TO X,16Ø-Y
85Ø NEXT I
86Ø END
9ØØ REM
91Ø REM **************
92Ø REM * DATA TABLE *
93Ø REM **************
94Ø REM
95Ø DATA 15,18
96Ø DATA 1ØØ,13Ø,9Ø,8Ø,1ØØ,21Ø,17Ø,6Ø,
    12Ø,3Ø,5Ø,4Ø,1ØØ,8Ø,9Ø
```

Listing 11-3. Line graph.

You may change the data plotted by changing the values given in line 960. The numbers on the vertical axis will change accordingly, but will still increment by 20. The increment for the vertical scale may be adjusted in line 520, but we will talk more about that soon. The first value in line 950 gives the number of data points to be plotted, and the second item indicates the horizontal spacing between plots. Both those numbers may be altered and the program will behave properly, except for the horizontal labels which remain fixed.

After you have played for a while, let's look at some of the more interesting parts of this program.

```
9Ø PRINT CHR$ (4);"BLOAD NUMBER TABLE,A$3ØØ"
1ØØ POKE 232,Ø: POKE 233,3
11Ø SCALE= 1
```

These lines load the shape table from disk, and tell the system where it was put. (See Chapter 10.)

```
300 DIM DP(NP%-1)
310 FOR I = 0 TO NP%-1
320 READ DP(I)
330 IF MAX < DP(I) THEN MAX = DP(I)
340 NEXT I
380 SCL = 150 / MAX
```

These lines read the data points into an array called DP and establish the maximum and the scaling factor. Thanks to line 380 the maximum point is plotted at Y-coordinate 150.

```
400 REM DRAW FRAME
420 HGR
430 HCOLOR= 6
440 HPLOT 24,0 TO 24,159 TO 278,159 TO 278,0 TO 24,0
```

Here a blue frame is drawn for the graph, and space is provided on the left side for the labels.

```
500 REM SCALE VERTICAL AXIS
520 HCOLOR= 3
530 FOR I = 20 TO MAX STEP 20
540 ROT= 16: DRAW 11 AT 24,160-I * SCL
550 ROT= 0
560 N$ = STR$ (I):NL = LEN (N$)
570 FOR J = 1 TO NL
580 DIGIT = VAL ( MID$ (N$,J,1)): IF
    DIGIT = 0 THEN DIGIT = 10
590 DRAW DIGIT AT (J-1) * 7,160-I * SCL
600 NEXT J
610 NEXT I
```

Here we scale the vertical by 20. If you wished to use a different scale, such as 10, you would change both occurrences of 20 in line 530 to 10. Line 540 rotates our little tic-mark (shape 11) and draws it across the axis. Line 550 returns the rotation to 0 and the section from lines 560 through 600 proceeds to take each digit of the label (I) and draws it on the Hi-Res screen by means of the shape table.

```
630 REM SCALE HORIZONTAL AXIS
650 ROT= 0
660 FOR I = 1 TO NP%-1
670 X = 24 + I * SP%
680 IF X < 280 THEN DRAW 11 AT X,159
690 NEXT I
700 VTAB (21)
```

```
71Ø PRINT "J F M A M J J A S O N D J F M
          A E A P A U U U E C O E A E A
          N B R R Y N L G P T V C N B R";
72Ø VTAB (2Ø)
```

This section labels the horizontal axis by a method which makes up in simplicity what it lacks in style. Lines 65Ø-69Ø draw the little tic-mark on the axis, and line 71Ø prints the labels in the four lines of text at the bottom of the Hi-Res page. Line 71Ø was actually designed after the graph had been plotted by using the "Escape Mode" editing features of the Apple. If you use HGR2 or leave Hi-Res space below the graph, you could label the horizontal axis in the same way we labeled the vertical.

```
75Ø REM * PLOT DATA POINTS *
78Ø HCOLOR= 7
79Ø HPLOT 24,16Ø-DP(Ø) * SCL
8ØØ FOR I = 1 TO NP%-1
81Ø X = 24 + I * SP%
82Ø Y = DP(I) * SCL
83Ø IF X > 279 THEN PRINT CHR$ (7);
   "CHART TOO WIDE": GOTO 86Ø: REM (END)
84Ø HPLOT TO X,16Ø-Y
85Ø NEXT I
```

The heart of the routine is in this short section. Line 79Ø plots the first point to establish an initial point for the line, then line 84Ø plots the rest of the points. The X-coordinate is the left side of the frame (24) plus the offset for each point. DP(4) will be plotted at 24 + 4 * 18, or 96.

The Y-coordinate is calculated by multiplying the data value by the scaling factor. It should seem curious that the Y-coordinate is not plotted directly, but is subtracted from 16Ø. Remember that the Apple thinks Ø is at the top of the screen and 16Ø at the bottom, so the quick little subtraction turns things around to put Ø on the bottom where you usually want it.

The 15 data points are stored in DP(Ø) through DP(14). This is why the FOR/NEXT loop established in line 8ØØ only goes to one less than the number of points (NP%—1).

Functional Plotting

Many times a graph may be drawn from a function or formula instead of data points. Listing 11-4 is a short example which graphs a couple of formulas. Try it out!

```
1Ø REM FUNCTION DEMO
2Ø REM
3Ø DEF FN A(X) = .Ø2 * X ^ 2 + 5
4Ø DEF FN B(X) = .ØØ1 * X ^ 3
```

```
5Ø CX = 14Ø:CY = 8Ø
6Ø REM
7Ø REM DRAW AXES
8Ø REM
9Ø HGR : HCOLOR= 3
1ØØ HPLOT CX-75,CY TO CX + 75,CY
11Ø HPLOT CX,CY-75 TO CX,CY + 75
12Ø REM
13Ø REM PLOT GRAPH
14Ø REM
15Ø FOR X = -4Ø TO 4Ø
16Ø Y = FN A(X)
17Ø HPLOT CX + X,CY-Y
18Ø Y = FN B(X)
19Ø HPLOT CX + X,CY-Y
2ØØ NEXT X
```

Listing 11-4. Formula plotting.

When the program runs, it draws a set of axes with the center at the point (14Ø,8Ø) as set in line 5Ø. Then it draws the functions defined in lines 3Ø and 4Ø. The Apple thinks that (Ø,Ø) is in the upper left corner of the screen, but we make it appear to be at (CX,CY) by adding those values before plotting the point. As in the other line graph example, we must SUBTRACT instead of adding the Y value from CY since Apple believes that Y increases from top to bottom, and we require Y to increase from bottom to top. Essentially, we flip the graph over by using -Y in place of Y.

The decimals in the function definition are scaling factors used to keep the otherwise large numbers (4Ø to the third power is 64,ØØØ) within range. Convenient labels may be added to this as in the previous example.

Graphing in Circles

Circle graphs, or pie charts, are a very popular form of presentation, but we do not know anyone who enjoys the tedious calculations required to draw them. Key in and run Listing 11-5.

```
1Ø REM PIE CHART DEMO
2Ø REM
3Ø REM ******************
4Ø REM * INITIALIZATION *
5Ø REM ******************
6Ø REM
7Ø DEF FN X(T) = 8Ø * COS (T)
8Ø DEF FN Y(T) = 7Ø * SIN (T)
9Ø REM
```

```
100 REM READ DATA
110 REM
120 READ NUM%
130 DIM DTA(NUM%)
140 FOR I = 1 TO NUM%
150 READ DTA(I)
160 TTL = TTL + DTA(I)
170 NEXT I
180 REM
190 REM ************
200 REM * DRAW PIE *
210 REM ************
220 REM
230 HGR : HCOLOR= 3
240 FOR T = 0 TO 1.57 STEP .01
250 X = FN X(T):Y = FN Y(T)
260 HPLOT X + 140,Y + 80
270 HPLOT -X + 140,Y + 80
280 HPLOT X + 140,-Y + 80
290 HPLOT -X + 140,-Y + 80
300 NEXT T
310 REM
320 REM ***********
330 REM * CUT PIE *
340 REM ***********
350 REM
360 FOR I = 1 TO NUM%
370 ACC = ACC + DTA(I)
380 R = (ACC / TTL) * 6.28
390 HPLOT 140,80 TO FN X(R) + 140,
    FN Y(R) + 80
400 NEXT I
410 END
500 REM
510 REM **************
520 REM * DATA TABLE *
530 REM **************
540 REM
550 DATA 5
560 DATA 231.4,287,310.84,340,390
```

Listing 11-5. Pie charts.

Line 550 gives the number of regions to be graphed, and line 560 lists the actual data used for each region. The regions are plotted clockwise from the right side of the circle. There are no real restrictions on the values, although attempting to plot a region for a negative value will give you a rather meaningless result.

After you have run the program a few times with your own data, we can go through it and hit the highlights.

```
7Ø DEF FN X(T) = 8Ø * COS (T)
8Ø DEF FN Y(T) = 7Ø * SIN (T)
```

The functions defined here are used to calculate the X and Y-coordinates of points on the circle. The values 8Ø and 7Ø define the radius of the circle (two radii?!?). The dots on the screen are slightly higher than they are wide, so if you let both numbers be 8Ø (as theoretically they should), the pie looks more like an egg.

```
12Ø READ NUM%
13Ø DIM DTA(NUM%)
14Ø FOR I = 1 TO NUM%
15Ø READ DTA(I)
16Ø TTL = TTL + DTA(I)
17Ø NEXT I
```

NUM% is the number of regions, and the data values are read into the array DTA. Since a total is required to calculate the relative size of each "slice," we accumulate the data values in TTL.

```
24Ø FOR T = Ø TO 1.57 STEP .Ø1
25Ø X = FN X(T):Y = FN Y(T)
26Ø HPLOT X + 14Ø,Y + 8Ø
27Ø HPLOT -X + 14Ø,Y + 8Ø
28Ø HPLOT X + 14Ø,-Y + 8Ø
29Ø HPLOT -X + 14Ø,-Y + 8Ø
3ØØ NEXT T
```

The circle is drawn using a little simple trigonometry. (There is really no such thing as simple trigonometry!) You will notice here (and also later in the program) that we place the center at (14Ø,8Ø) by adding 14Ø to each X and 8Ø to each Y as we plot.

The real question is, why does T go from Ø to 1.57? The SIN and COS functions we use measure a circle in radians instead of degrees. There are roughly 6.28 radians in a full circle (3.14 in a semicircle), and we use the symmetry of the circle so that we only need to calculate the points for one-quarter of the circle. The result of all this is that we go one-quarter of the way to 6.28, and one-quarter of 6.28 is the elusive 1.57.

```
36Ø FOR I = 1 TO NUM%
37Ø ACC = ACC + DTA(I)
38Ø R = (ACC / TTL) * 6.28
39Ø HPLOT 14Ø,8Ø TO FN X(R) + 14Ø,
    FN Y(R) + 8Ø
4ØØ NEXT I
41Ø END
```

As we draw the wedges of the pie, our progress around the circle is proportional to the amount of the data graphed so far. That is, if we are one-third of the way around the circle, then the accumulated amount of data already graphed is one-third of the total to be graphed. So, as we go around, line 37Ø accumulates the graphed data in ACC, and line 38Ø calculates the fraction of a full rotation. (Remember, in radian measure 6.28 represents a full rotation.)

Line 39Ø draws the line from the center of the circle (14Ø,8Ø) to the point on the circle at the correct rotation.

Round It Out

You have seen several fairly powerful schemes for graphing. They all produce a reasonable result on the screen, but what about getting a printout of the graph? As with most any programming problem, where there's a will there's a way, but the most practical solution lies in printer hardware. There are several items presently on the market which allow you to "dump" the Hi-Res screen to a matrix printer, such as "GRAFTRAX*" for Epson printers, or the "GRAPPLER**" interface which works for most dot-matrix machines. If you have, or acquire such an item, it comes with instructions for dumping the Hi-Res screen.

The programs presented here are designed to let you plot your own data with a minimum of modifications. After you have used these routines for a time they will become clearer to you, and improvements will begin to suggest themselves. Down that path lies hours of both creativity and frustration. Have fun!

Vocabulary

Bar chart

Circle graph

Line graph

Pie chart

*Graftrax is a trademark of Epson of America.
**Grappler is a trademark of Microtech, Inc.

Exercises

1. Use Listing 11-1 to do a bar graph showing a profit of 7,250 in January, 4,173 in February, 5,010 in March, 6,379 in April, 5,703 in May, and 6,533 in June.
2. Use the number table (Listing 11-2) and Listing 11-3 to graph the following values: 17.9, 26.4, 25.8, 32, 20.5, 10.9, 8.05, 6.1, 9.0, 12.7, and 16.
3. Use the number table (Listing 11-2) and Listing 11-4 to graph the function $Y = X^3 - 8X^2 - 17X + 10$. Label each axis.
4. Use the number table to create a pie chart with the following data: 150, 470, 173, 217, 301, 522. Label the resulting graph.

12

Byte-Move Shapes

Objectives

After reading Chapter 12 you should be able to:

- Use byte-move to perform a stationary and a vertical animation.
- Draw and digitize the seven horizontal separations of a byte-move figure.
- Animate a simple byte-move figure across the screen.

The standard Apple shape is a fairly useful construct, particularly for the BASIC programmer because it lets the user use a number of fast machine-level routines via the APPLESOFT shape commands (DRAW, XDRAW, SCALE, and others). Unfortunately, those routines are not fast enough for arcade type animation, so byte-move graphics are often used to gain extra speed. To be honest, professional byte-move animation is written in machine language because BASIC is too slow to gain much advantage, whereas "look-up tables" are something that a machine-level program handles very well. These examples in BASIC serve to give you the "basic" idea behind byte-move, so if you are so inclined and endowed you can write these strategies into your machine-level program.

It is important to realize how byte-move shapes differ from standard shapes. As you saw in Chapter 1Ø, a shape is defined by a series of instructions (vectors). For example, plot-up, skip-left, skip-left, plot-down, and so on. Each time the shape is plotted, the drawing routines follow those directions and reconstruct the shape. Since shapes can easily run to a hundred instructions or more, the vector method can lead to a time delay as each vector is reprocessed and reinterpreted.

A byte-move shape can be thought of as being pre-defined and pre-drawn someplace in the computer's memory. Plotting a byte-move shape is a matter of moving the pre-drawn figure into the screen memory area, just as you might use pre-printed rub-on transfers for lettering. In this way, a byte-move shape saves time since it does not have to be regenerated each time it is used, but has only to be moved from another location—prefab graphics.

We are getting ahead of ourselves. Listing 12–1 gives a short example of animation using the byte-move process. Type it in and try it. You may omit the REM statements if you wish.

```
10 REM INITIALIZE Y
20 REM COORDINATES
30 REM
40 Y1% = 1:Y2% = 2:Y3% = 3:Y4% = 4:
   Y5% = 5:Y6% = 6:Y7% = 7
50 REM
60 REM READ DATA FOR FIGURE
70 REM
80 FOR I = 1 TO 4: REM 4 FRAMES
90 FOR J = 1 TO 7: REM 7 BYTES PER
   FRAME
100 READ V%(I,J)
110 NEXT J,I
120 REM
130 REM INITIALIZE ADDRESSES
140 REM OF Y COORDINATES
150 REM
160 Y%(1) = 8192:Y%(2) = 9216:Y%(3) = 10240:
    Y%(4) = 11264:Y%(5) = 12288:Y%(6) = 13312:
    Y%(7) = 14336
170 HGR
180 REM
190 REM POKE THE FOUR FRAMES
200 REM
210 FOR I = 1 TO 4
220 POKE Y%(Y1%),V%(I,1):
    POKE Y%(Y7%),V%(I,7)
230 POKE Y%(Y2%),V%(I,2):
    POKE Y%(Y6%),V%(I,6)
240 POKE Y%(Y3%),V%(I,3):
    POKE Y%(Y5%),V%(I,5):
    POKE Y%(Y4%),V%(I,4)
250 NEXT
260 GOTO 210: REM START AGAIN
270 REM
280 REM DATA FOR THE FOUR FRAMES
```

```
290 REM
300 DATA 1,2,4,8,16,32,64
310 DATA 8,8,8,8,8,8,8
320 DATA 64,32,16,8,4,2,1
330 DATA 0,0,0,127,0,0,0
```

Listing 12-1.

When the program is running correctly, you will see what could pass for an airplane propeller spinning in the corner of your screen. The effect is accomplished by cycling through four different frames quickly enough to fool your eyes and brain into seeing a continuous motion—the POKE is quicker than the eye! The four frames are diagrammed in Figure 12-1, along with the bit patterns and decimal values required to generate them.

SCREEN PATTERN	BINARY VALUE	DECIMAL
X------	0000 0001	1
-X-----	0000 0010	2
--X----	0000 0100	4
---X---	0000 1000	8
----X--	0001 0000	16
-----X-	0010 0000	32
------X	0100 0000	64

FRAME #1

SCREEN PATTERN	BINARY VALUE	DECIMAL
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8
---X---	0000 1000	8

FRAME #2

SCREEN PATTERN	BINARY VALUE	DECIMAL
------X	0100 0000	64
-----X-	0010 0000	32
----X--	0001 0000	16
---X---	0000 1000	8
--X----	0000 0100	4
-X-----	0000 0010	2
X------	0000 0001	1

FRAME #3

SCREEN PATTERN	BINARY VALUE	DECIMAL
------	0000 0000	0
------	0000 0000	0
------	0000 0000	0
XXXXXX	0111 1111	127
------	0000 0000	0
------	0000 0000	0
------	0000 0000	0

FRAME #4

Figure 12-1.

The values for each frame are the ones used in the four DATA statements in the program.

When the program is run, there is a delay before anything seems to happen. This delay is a characteristic of byte-move programs, and is

caused by the need to initialize a number of variables and arrays before beginning the animation. In Listing 12-1, line 40 sets the seven Y coordinates used in the figure, and line 160 assigns the corresponding memory address to each of those coordinates. Lines 80 through 110 initialize an array that contains seven data values for each of the four versions of the propeller.

The actual animating is done in lines 210 through 250 which POKE the values for each frame into Hi-Res memory. The variables and arrays used in the POKEs tend to obscure some of the mechanics of the program, but they also enhance the execution, since it is much faster for the machine to look up the value of Y%(4) stored in memory than to generate the corresponding value of 12288.

In general it is faster to look something up in a table than to generate it each time it is needed.

Line 220 POKEs the first and seventh bytes of the figure [V%(I,1) and V%(I,7)], 230 the second and sixth, and 240 the third, fifth, and fourth bytes. POKEing the values in that weird order improves the resulting effect, but you might try changing the order to POKE them sequentially just to see what happens.

Another reason for the variables is to make it easy to modify this program so that the figure will plot at different Y coordinates on the screen. The array Y% will contain the starting addresses for each of the 192 screen lines, and Y1% through Y7% will be the coordinates for the seven lines used in the shape.

The propeller is an example of stationary animation. That is, animation that takes place within a fixed region, and the different frames of that region are alternated to simulate movement. There are many applications for stationary animation. The APPLEVISION program on your DOS 3.3 master is an excellent example, and so is a Hi-Res scoreboard for a game. (The ten different digits compose ten possible frames for display.)

Drop the Prop

Most shapes used in a game are required to move about the screen. We will now alter our program to allow for this. With Listing 12-1 still in memory, type in the following changes.

```
160 GOSUB 1000: REM CALC ADDRESSES
245 Y1% = Y1% + 1: Y2% = Y2% + 1:
    Y3% = Y3% + 1: Y4% = Y4% + 1:
    Y5% = Y5% + 1: Y6% = Y6% + 1:
    Y7% = Y7% + 1
246 IF Y7% > 192 THEN END
1000 REM
```

```
1010 REM CALCULATE Y
1020 REM COORDINATES
1030 REM
1040 DIM Y%(192)
1050 FOR I = 1 TO 185 STEP 8: READ SA%
1060 FOR J = 0 TO 7:Y%(I + J) = SA% +
     J * 1024
1070 NEXT J,I
1080 DATA 8192,8320,8448,8576,8704,8832,
     8960,9088
1090 DATA 8232,8360,8488,8616,8744,8872,
     9000,9128
1100 DATA 8272,8400,8528,8656,8784,8912,
     9040,9168
1110 RETURN
```

Listing 12-2.

Lines 1000 through 1110 calculate the starting addresses for each of the 192 screen lines by taking the base address of the box that the line lies in and adding the position address to it. That position address is calculated as a multiple of 1024. If that algorithm escapes you, try reviewing the portion of Chapter 8 where we talked about calculating addresses for Hi-Res memory.

Line 245 is executed after each frame is displayed, and its purpose is to increment each of the seven Y coordinates so that the prop will be plotted one line lower the next time.

Run the modified program. You will see the propeller dropping down the left edge of the screen as it spins. You will also see a trail of garbage left behind as the shape progresses. Most of each frame is wiped out when the next one plots over it, but since we are moving down between frames, the top line of each frame remains on the screen to haunt you.

When moving a shape using byte-move, it is often neccesary to make special provisions for erasing the left-overs.

We all know that your first try never runs perfectly anyway, and the problem is easily remedied by adding:

```
215 POKE Y%(Y0%),0
244 Y0% = Y1%
```

Line 244 stores the Y-coordinate for the old top line as Y0%, and line 215 POKEs a zero there to erase that byte.

Getting It Across

So far, you have performed a stationary and a vertical animation using

byte-move techniques. Now we take up the problem of horizontal animation, and a problem it turns out to be indeed, for it becomes about seven times as complex as a vertical animation. For that reason we will confine our byte-move examples to the animation of simple figures.

Lucky Seven

Seven seems to be the magic number for horizontal byte-move, and by moving a single dot across the screen a little way, we can show you why. If you plan to turn on the dot in the extreme upper-left corner of the Hi-Res screen, you need to address yourself to the byte at 8192. The byte contains seven bits, and you want to turn on only the left-most, so you need the dot pattern:

```
X------
```

which corresponds to:

```
!0000 0001 or 1
```

so you may turn the dot on by typing:

```
HGR
POKE 8192,1
```

Next, you want the second dot on, then the third, fourth, fifth, sixth, and seventh, in turn. Figure 12-2 sets out the desired dot and bit patterns, and then the corresponding values.

SCREEN PATTERN	BINARY VALUE	DECIMAL
X------	0000 0001	1
-X-----	0000 0010	2
--X----	0000 0100	4
---X---	0000 1000	8
----X--	0001 0000	16
-----X-	0010 0000	32
------X	0100 0000	64

Figure 12-2.

From BASIC type the following:

```
HGR
POKE 8192,1
POKE 8192,2
POKE 8192,4
POKE 8192,8
POKE 8192,16
```

```
POKE 8192,32
POKE 8192,64
```

The dot will move to the right, and the last statement will leave the dot at the right edge of byte number 8192. To continue the journey, the dot must move to the first dot of byte number 8193, and then continue across that byte. To accomplish that, you can type the same series of values, but use 8193 as the address:

```
POKE 8193,1
POKE 8193,2
POKE 8193,4
POKE 8193,8
POKE 8193,16
POKE 8193,32
POKE 8193,64
```

This pattern will repeat until you get through address 8231, or you get tired of it, whichever comes first.

As you can see, each position across the byte requires a different value, and there are seven positions across the byte, so seven becomes a very important number. You can also see that the last dot in every byte you travelled across was left on. As we mentioned before, some arrangement must be made to erase the last value before moving on to the next byte. POKEing a zero as the final value in each series will erase that left-over dot.

To move the dot across byte number 16 in that row (the row starts with byte number Ø), you would find the address by adding the offset, 16, to the base address, 8192 (did everybody get 82Ø8?), and then POKE the series of values using that address.

There is a nice relationship between the X-coordinate of a point and the offset and value needed to refer to it in byte-move. Consider the seven values used to move the dot across the byte as seven versions of the dot, and number them zero through six. If you wish to turn on a particular dot across the row, you need to determine which version of the dot to POKE, and the offset from the beginning of the row.

For example, let's turn on the dot in the first row which has an X-coordinate of 75. Divide the coordinate by seven to find the quotient (the result of the division) and remainder. Seven goes into 75 ten times (remember your g'zintas?) with five remaining. The quotient (1Ø) is the offset, and the remainder (5) is the number of the separation (versions of the dot) to be used. Therefore, the address is 82Ø2 (8192 + 1Ø), and the value is 32. So,

```
POKE 82Ø2,32
```

and you will turn on the dot having 75 as its X-coordinate.

For another example, turn on the dot with X-coordinate 100 and Y-coordinate 80. Screen line 80 is at the top of box number 10, so it has a starting address of 8488. (See the Hi-Res memory map in Appendix 3.) Dividing 100 (the X-coordinate) by seven gives you a quotient of fourteen, and a remainder of two. Value number 2 is four, so

```
POKE 8488 + 14,4
```

to turn the dot on. In our programming examples, we will represent the quotient by Q%, and the remainder by R%.

Byting Off More

In the next few examples you will animate a line across the width of the screen. The line is to be seven dots long, so initially it will fit nicely in one byte. Run this simple example.

```
10 HGR
20 FOR L = 8192 TO 8231
30 POKE L,127: REM POKE NEW BYTE
40 POKE L-1,0: REM ERASE OLD BYTE
50 FOR I = 1 TO 50: NEXT I
60 NEXT L
```

Listing 12-3.

The program moves a line across the top of the screen quickly, even with the delay at line 50, but it still has a drawback fatal to any game: the animation is jittery instead of smooth. Each cycle of the loop POKEs the line one byte further on, equivalent to moving seven dots at a time, and that is too large a step. The solution is to move the line only one dot at a time, though that is easier said than done.

Imagine that you are looking out a window which is seven dots wide, and a snake (also seven dots worth) crawls across the field of vision from left to right. At first you will see only one dot of the snake at the left side of the window, then two dots, then three, four, five, six, and finally all seven dots as the snake is completely within the window. In computer terms, the window you are looking through is one byte of the Hi-Res screen, and the snake is the line progressing across it.

Meanwhile, Back at the Snake...

We left the snake completely within one byte. As it crawls onward, it enters the next window in the same way it entered the first, and at the same time it leaves the first window one dot at a time until the first byte is empty and the second is filled. This sequence is shown in Figure 12-3.

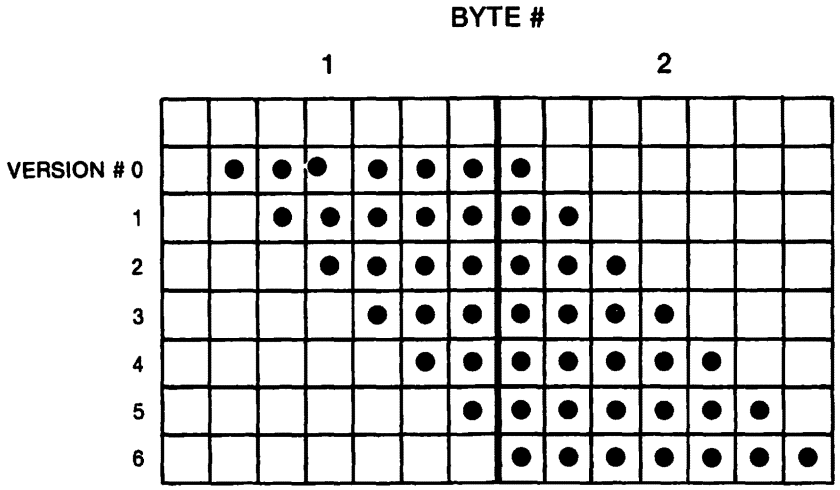


Figure 12-3.

If you take the time to calculate the values needed to produce the seven versions, you should arrive at:

BYTE #1		BYTE #2	
DEC VALUE	DOT PATTERN	DOT PATTERN	DEC VALUE
126	-XXXXXX	X------	1
124	--XXXXX	XX-----	3
120	---XXXX	XXX----	7
112	----XXX	XXXX---	15
96	-----XX	XXXXX--	31
64	------X	XXXXXX-	63
0	-------	XXXXXXX	127

What you have in this example is a two-byte animation. Even though the figure appears to be only one byte wide, in six of the seven possible placements, it requires two bytes to represent the line. You should consider each of the bytes in a figure to have its own set of seven separations.

It is another characteristic of byte-move that any figure requires seven separations for horizontal movement and, further, that each figure takes one byte more than you would expect.

The line which fits in one byte actually requires a two-byte definition. A four-byte figure will require five bytes for all the separations. The only exception is a figure one dot wide, where all seven versions will fit into one byte.

You can manually move the line from BASIC by typing

```
HGR
```

```
POKE 8192,126: POKE 8193,1
POKE 8192,124: POKE 8193,3
POKE 8192,12Ø: POKE 8193,7
POKE 8192,112: POKE 8193,15
POKE 8192,96: POKE 8193,31
POKE 8192,64: POKE 8193,63
POKE 8192,Ø: POKE 8193,127
```

Listing 12–4 gives you a program that will do essentially what you did with the series of POKEs.

```
1Ø DIM A%(28Ø): REM 28Ø X-COORDINATES
2Ø REM
3Ø REM READ THE VALUES FOR
4Ø REM THE 7 PAIRS OF FRAMES
5Ø REM
6Ø FOR I = Ø TO 6
7Ø READ T%(I),H%(I)
8Ø NEXT I
9Ø REM
1ØØ HGR
11Ø REM
12Ø REM INITIALIZE THE TABLE
13Ø REM OF ADDRESSES
14Ø REM
15Ø J = Ø
16Ø FOR I = 8192 TO 8231
17Ø A%(J) = I:J = J + 1
18Ø NEXT
19Ø REM
2ØØ REM PLOT THE LINE AT
21Ø REM EACH X-COORDINATE
22Ø REM
23Ø FOR X = 1 TO 28Ø
24Ø Q% = INT (X / 7)
25Ø R% = X-(7 * Q%)
26Ø C% = Q% + 1
27Ø POKE A%(Q%),T%(R%): POKE
    A%(C%),H%(R%)
28Ø NEXT X
29Ø END
294 REM
295 REM DATA TABLE
296 REM
3ØØ DATA 126,1,124,3,12Ø,7,112,15
31Ø DATA 96,31,64,63,Ø,127
```

Listing 12–4.

Again, we use variables to make the program obscure. The values for the line are read into arrays T% (for tail) and H% (head) in lines 30 to 80. Array A% contains the addresses for each of the 40 bytes across the top of the screen, and is initialized in lines 120 through 180. The loop in lines 230 to 280 plots the line at every X-coordinate across the screen, but lines 240, 250, and 260 merit more study.

Their purpose is to determine which pair of bytes is being used, and which of the seven pairs of values need to be POKEd. They do this by using the quotient and remainder. For example, when the X-coordinate is 42, seven g'zinta 42 six times, with zero left over. So you need to POKE the sixth (and seventh) byte, with the zeroth pair of values.

When designing a byte-move shape, you will need to sketch and digitize all seven separations of the figure as we did for the line. That data gets stored in a table for use later in the program. As you can see, even as simple a figure as the line required seven different versions, and significant preparation to animate horizontally, but the resultant animation is as smooth as you could wish for, even though it is a bit slow.

Vocabulary

Byte-move

Quotient

Remainder

Separation

Exercises

1. Design Hi-Res figures for the digits zero through nine, and write a BASIC program to place them into a region of the Hi-Res screen in ascending order.
2. Alter the program written for Exercise 1 to move each digit down one screen line from the previous one.
3. Draw and digitize the seven versions of the numeral "5" needed to animate it horizontally.
4. Write the code necessary to animate the numeral "5" across the screen.

13

Advanced Moves

Objectives

After reading Chapter 13 you should be able to:

- Use pre-calculated tables to help enhance your animation.
- Animate stationary figures using partial modification.
- Develop pre-shifts for shapes and use them for animation.

We commend you for getting this far in your study of Apple II graphics. (No fair if you just skimmed to this point!) You have progressed from a knowledge of BASIC through the land of binary and hex numbers, on through the memory map maze, and to the realm of Low-Res graphics. From there you progressed to the world of Hi-Res where you investigated color, shapes, shape-table animation, and byte-move animation.

You have seen most of the major schemes currently used to produce Hi-Res graphics, and we think you are ready for a few advanced techniques. In this chapter we will look at three different methods which can increase the speed and efficiency of your graphics. Though the ideas discussed may be applied in either BASIC or machine-code, the examples are presented in BASIC for simplicity. The amount of benefit derived from each technique will depend on the particular application, and may vary from a lot to none at all (or worse). However, since a great amount of time spent programming any game is devoted to cleaning up the graphics, sometimes you have to be satisfied with several small improvements.

We will start with the idea of partial modification, where instead of redrawing the entire figure each time, you only plot those bytes which have changed from the previous figure. A good illustration of this idea is

provided by a scoreboard in Hi-Res; the digits keep changing in a predictable manner, and you can use that fact to shorten your code. Enter and run the following program (you may omit the REM statements if you wish).

```
10 REM PARTIAL MOD.
20 REM
30 HGR
40 REM
50 REM POKE EIGHT
60 REM
70 POKE 8192,60: POKE 9216,66:
   POKE 10240,66: POKE 11264,60
80 POKE 12288,66: POKE 13312,66:
   POKE 14336,60: POKE 15360,0
90 VTAB (24): PRINT "PRESS A KEY ":
   GET R$
100 REM
110 REM POKE NINE
120 REM
130 POKE 8192,60: POKE 9216,66:
    POKE 10240,66: POKE 11264,60
140 POKE 12288,64: POKE 13312,64:
    POKE 14336,60: POKE 15360,0
150 VTAB (24): PRINT "PRESS A KEY" :
    GET R$
160 REM
170 REM POKE ZERO
180 REM
190 POKE 8192,60: POKE 9216,66:
    POKE 10240,66: POKE 11264,66
200 POKE 12288,66: POKE 13312,66:
    POKE 14336,60: POKE 15360,0
210 VTAB (24): PRINT "PRESS A KEY":
    GET R$
220 GOTO 70
```

Listing 13-1.

When you have everything keyed in correctly, you will see the digits eight, nine, and zero cycle on the Hi-Res screen. This was easily done by POKEing the appropriate dot patterns into screen memory.

For example, the numerals eight and nine are composed of eight rows of dots which correspond to byte values as shown below.

DOT PATTERN	BIT PATTERN	DECIMAL VALUE	DOT PATTERN	BIT PATTERN	DECIMAL VALUE
--XXXX-	Ø011 11ØØ	6Ø	--XXXX-	Ø011 11ØØ	6Ø
-X----X	Ø1ØØ ØØ1Ø	66	-X----X	Ø1ØØ Ø1ØØ	66
-X----X	Ø1ØØ ØØ1Ø	66	-X----X	Ø1ØØ ØØ1Ø	66
--XXXX-	Ø011 11ØØ	6Ø	--XXXX-	Ø011 11ØØ	6Ø
-X----X	Ø1ØØ ØØ1Ø	66	------X	Ø1ØØ ØØ1Ø	66
-X----X	Ø1ØØ ØØ1Ø	66	------X	Ø1ØØ ØØ1Ø	66
--XXXX-	Ø011 11ØØ	6Ø	--XXXX-	Ø011 11ØØ	6Ø
-------	ØØØØ ØØØØ	Ø	-------	ØØØØ ØØØØ	Ø

Figure 13-1.

Now, if you take a minute to compare the values used for eight with those used for nine, you should notice that all except two of the values are the same. So the question naturally arises, "Since nine always follows eight, why should I POKE all the values for the nine, when six of them are the same as before?!?" Funny you should ask!

That is the idea behind partial modification—alter the existing figure instead of totally replacing it. Type these lines to change the previous listing.

```
13Ø REM
14Ø POKE 12288,64: POKE 13312,64
19Ø POKE 11264,66
2ØØ POKE 12288,66: POKE 13312,66
```

Now lines 13Ø and 14Ø will POKE only the changes needed to turn the eight into the nine, and similarly lines 19Ø and 2ØØ POKE the changes required to turn nine into zero. When you run the program you will not see much difference, but you will have the satisfaction of knowing that your code is more efficient.

Granted, the time savings is insignificant in this example, but if you are trying to animate 15Ø bytes worth of Zylon spaceship, or what have you, partial modification could potentially save a great deal of time.

Pre-calculation

The next topic to consider is pre-calculation. When a figure is moving

around the screen, there are many calculations to be made: the shape's X and Y-coordinates, perhaps the address corresponding to those coordinates, and if you are animating using byte-move, which of the seven versions of the shape is to be used at each coordinate.

Sometimes the figuring can be done "on the fly," such as after the shape is drawn on the screen and before it is erased to be moved. This increases the time the object is on the screen. This increases the ratio of the display time to the erase time, which in turn reduces flicker. However, calculating all of this within the animation routine will slow it down because arithmetic operations tend to gobble up relatively large amounts of processor time—remember, it is usually faster to look it up than to figure it out.

As an alternative, it is possible to compute the path of an object before it starts, and store each of the coordinates in a table (BASIC calls them arrays). It is usually faster to look a number up in a table (especially when using machine code) than it is to compute it on the spot. The game THRESHOLD uses tables, byte-move, and pre-calculation in its animation.

In Chapter 12 we used byte-move techniques to move a line across the screen. If you still have that program lying around on a disk somewhere, go get it because we are about to modify it. The complete listing is given below, but if you have the old program you can avoid having to type most of it in again.

```
10 DIM A%(280): REM 280 X-COORDINATES
20 REM
30 REM READ THE VALUES FOR
40 REM THE 7 PAIRS OF FRAMES
50 REM
60 FOR I = 0 TO 6
70 READ T%(I),H%(I)
80 NEXT I
90 REM
100 REM
110 REM
120 REM INITIALIZE THE TABLE
130 REM OF ADDRESSES
140 REM
150 J = 0
160 FOR I = 14336 TO 14374
170 A%(J) = I:J = J + 1
180 NEXT
190 REM
200 REM PRE CALC.
205 REM
210 DIM Q%(280),R%(280)
220 FOR X = 1 TO 280
230 Q%(X) = X / 7:R%(X) = X-Q%(X) * 7
```

```
240 NEXT X
250 HGR : REM SET GRAPHICS
254 REM
255 REM HERE GOES!!
256 REM
260 FOR X = 1 TO 280
270 POKE A%(Q%(X)),T%(R%(X)):
    POKE A%(Q%(X) + 1),H%(R%(X))
280 NEXT X
290 END
294 REM
295 REM DATA TABLE
296 REM
300 DATA 126,1,124,3,120,7,112,15
310 DATA 96,31,64,63,0,127
```

Listing 13-2.

Lines 100, 160, and 200 through 270 are the only new modifications.

The arrays Q% and R% hold the quotients and remainders for each of the 280 X-coordinates. Dividing the X value by 7 gives you the offset (0-39) used to address the correct byte across the screen, and also which of the seven versions of the figure should be used (Remember?).

Lines 160-180 fill array A% with the addresses for each of the 40 bytes across the screen line, and lines 220-240 calculate the 280 quotients and remainders. Finally, line 250 turns on Hi-Res, and line 270 does the actual POKEing.

Sorry about the compound indexing [A%(Q%(X))], but it could not be helped. X is the coordinate number, so Q%(X) is the quotient belonging to that coordinate, and A%(Q%(X)) is the address determined by that quotient.

This new version of the program runs the line across the screen in seven seconds, as opposed to nine in the earlier one. Now, seven seconds is still pretty slow (blame BASIC), but pre-calculating did result in a significant improvement (22 percent, since you asked.)

Pre-shifting

And now, on to perhaps the most elegant of the techniques: pre-shifting. We are going to animate using a shape table, so take a couple of minutes now to enter it. From BASIC type:

```
CALL -151
```

```
300: 02 00 06 00 45 00 3F 3F
308: 3F 3F 3F 3F 08 2D 2D 2D
310: 2D 2D 2D 18 3F 3F 3F 3F
318: 3F 3F 08 2D 2D 2D 2D 2D
320: 2D 18 3F 3F 3F 3F 3F 3F
328: 08 2D 2D 2D 2D 2D 2D 18
330: 3F 3F 3F 3F 3F 3F 08 2D
338: 2D 2D 2D 2D 2D 18 3F 3F
340: 3F 3F 3F 3F 00 24 24 24
348: 24 DF DB DB DB 06 36 36
350: 36 36 00 00 00 00 00 00

3D0G
BSAVE SQUARE,A$300,L$53
```

With the table still in memory, let's find out what we actually have there. Type:

```
POKE 232,0: POKE 233,3
HCOLOR = 3:ROT = 0:SCALE = 1:HGR
DRAW 1 AT 50,50
```

The first line tells APPLESOFT where the table is stored ($0300) in lo-byte/hi-byte form (00 and 03). The second line sets all the parameters, and the third draws the first shape, a rectangle, at 50,50 on the Hi-Res screen.

The second shape in the table is the horizontal pre-shift (the what!??) of the original rectangle. To explain, imagine that you shift the rectangle one place to the right. The bulk of the figure is unchanged; there is a single line added to the right side, and one deleted from the left side. Pre-shifting, like partial modification, is a way to process only that portion of the figure which changes while leaving the rest alone.

Now type:

```
DRAW 2 AT 51,60
```

to draw the pre-shift below the rectangle. Notice that the pre-shift has a single line to the right of the rectangle, and another that lines up with the left side of the rectangle. To affect the modification, we will XDRAW the pre-shift on top of the rectangle.

Let us digress for a moment. When used to superimpose one figure on another, XDRAW has the effect of comparing corresponding dots of the two shapes and forming a resultant figure from them. The resultant dot is on if either one of the original dots were on, but not both. The chart below summarizes the results from the four possibilities.

Dot#1	ON	ON	OFF	OFF
#2	ON	OFF	ON	OFF
	---	---	---	---
Result	OFF	ON	ON	OFF

So, when the pre-shift is XDRAWn over the rectangle, the dots on the left side of both figures are on, so that whole row will be turned off. But the right side of the pre-shift is one row beyond the rectangle, and since only the pre-shift dots are on, the result will be to turn that row of dots on. But enough words, let's try it! From BASIC type:

```
XDRAW 2 AT 51,5Ø
XDRAW 2 AT 52,5Ø
XDRAW 2 AT 53,5Ø
XDRAW 2 AT 54,5Ø
XDRAW 2 AT 54,5Ø
XDRAW 2 AT 53,5Ø
```

The rectangle will move to the right, and then back to the left. If you are surprised by the repeated XDRAWs at 54, remember that two consecutive XDRAWs always cancel out. The first XDRAW moves the rectangle right, and the second cancels it out to move the rectangle back to the left. Play with XDRAWing this figure manually until you can move it around comfortably.

Listing 13–3 uses this idea to move the rectangle across the screen.

```
1Ø REM PRE-SHIFT
2Ø REM
3Ø D$ = CHR$ (4)
4Ø PRINT D$ "BLOAD SQUARE"
5Ø POKE 232,Ø: POKE 233,3
6Ø HCOLOR= 3: ROT= Ø: SCALE= 1
7Ø HGR
8Ø DRAW 1 AT 2Ø,1ØØ
9Ø FOR I = 21 TO 275
1ØØ XDRAW 2 AT I,1ØØ
11Ø NEXT I
```

Listing 13–3.

There are two very pleasant surprises with this program. The first is that it is short and simple, and the second is that it moves the figure quickly and with very little flicker!

You might wish to compare this method with the earlier one of XDRAWing the shape on and off the screen. To do so, just make the following changes:

```
8Ø FOR I= 2Ø TO 275
9Ø XDRAW 2 AT I,1ØØ
1ØØ XDRAW 2 AT I,1ØØ
```

Pre-shifting can be used for both horizontal and vertical movement. To determine the pre-shift for any figure, just XDRAW it once, shift in the

desired direction, and XDRAW it again. To demonstrate, let's find the vertical pre-shift of our rectangle. From BASIC type:

```
HGR
XDRAW 1 AT 5Ø,5Ø
XDRAW 1 AT 5Ø,49
```

The figure remaining on the screen is the vertical pre-shift of the rectangle. To use it, you need to include it in a shape table and then XDRAW it over the rectangle. Doing this will make the rectangle appear to move vertically. Creating a shape out of the pre-shift may still be a small problem, but several of the Hi-Res editors are capable of that.

Pre-shifting may also be used with byte-move shapes. To do this, the full figure is first drawn on the screen, and from then on pre-shifts are used to move it. As a consequence, there are seven versions of the pre-shift instead of seven versions of the figure, and the proper pre-shift is selected by examining the remainder.

Pre-shifting with byte-move requires that you "Exclusive Or" (EOR) the bytes in the pre-shift over the bytes on the screen. Remember, an "Exclusive Or" operation compares two bits and gives a positive (ON) result if either one of the originals was ON, but not both. Otherwise, the result is negative (OFF). The XDRAW command does exactly that with shape table figures, but since byte-move does not use a shape table, you must arrive at your own EOR routine. This is a pain from BASIC, but machine language, where byte-move is most advantageous anyway, has its own EOR command so the routine is fairly easy to write.

Back in the Paddle Again

Just for fun, let's modify Listing 13-3 further to use paddle Ø to control the rectangle.

```
8Ø DRAW 1 AT 2Ø,1ØØ
9Ø X = 2Ø:XO = 2Ø
1ØØ OD = 1
11Ø REM
12Ø IF PDL (Ø) < 9Ø THEN X = X-1:
    ND = Ø: GOTO 14Ø
13Ø IF PDL (Ø) > 15Ø THEN X = X + 1:
    ND = 1
14Ø IF X < 2Ø OR X > 275 THEN X = XO
15Ø IF X = XO GOTO 11Ø
16Ø IF ND < > OD THEN XDRAW 2 AT
    XO,1ØØ
17Ø XDRAW 2 AT X,1ØØ:XO = X:OD = ND
18Ø GOTO 11Ø
```

Lines 12Ø and 13Ø test the value of PDL(Ø) and decrement or increment the X-coordinate as needed. Line 14Ø keeps everything within range, and line 15Ø skips the XDRAW statements if there is no movement of the square—this helps to avoid flicker.

In line 16Ø, if the old direction (OD) is different from the new direction (ND) then the extra XDRAW is done. (Remember that series of XDRAW statements above?) Line 17Ø does the regular XDRAW and sets the old values to the new values.

With these changes in effect, the rectangle will respond to paddle control.

When you add partial modification, pre-shifting, and pre-calculation to your repertoire, you give the final polish to your animation skills. That is about all there is to know about designing and animating figures, and in the next chapter we address a slightly different topic: collision detection.

*** One extra tid-bit for the Apple IIe***

One annoying source of flicker in animation stems from changing the data being displayed while it is being scanned. The electron beam in your monitor or TV sweeps over the entire screen sixty times every second, much as a searchlight will sweep across the sky. The trick is to change the display data when the beam is not "looking." Over one-quarter of the time in each sweep is spent getting the beam from the bottom back to the top of the screen so it can begin the next cycle. This is called Vertical Blanking. Since nothing new can be displayed during this interval (about one-2ØØth of a second), this is the best time to switch the display data. With the Apple II+ there was no way to tell when the vertical blanking was occurring, but the IIe has a special Vertical Blanking Location (VBL) at 49177 ($CØ19) which signals the vertical blanking interval to anyone who cares to look. When the vertical blanking interval begins, the value in the VBL drops to less than 128, and this value may be read using a PEEK statement.

Let's suppose that the routine to animate the figure by changing the displayed data begins at line 1ØØØ, then line 1ØØØ could read:

```
1ØØØ IF PEEK (49177) > 127 GOTO 1ØØØ
1Ø1Ø Routine to POKE in new values
      .
      .
      .
```

Line 1ØØØ will delay the changing of display values until the vertical blanking period begins, at which time it will drop through to line 1Ø1Ø and begin changing.

The VBL is still too new to fairly evaluate its usefulness, but it is getting mixed reviews from some of the current game programmers. Although it is nice to change the display data during the vertical blanking interval, doing

so may slow the animation since movement is always delayed until the next vertical blanking interval. Having to wait for less than one-sixtieth of a second may not seem like much to you, but to your Apple it is quite a long time. We will just have to wait and see what new techniques are developed for the Apple IIe.

Vocabulary

Exclusive OR (EOR)

Partial Modification

Pre-Calculation

Pre-Shift

Vertical Blanking

Vertical Blanking Location (VBL)

Exercises

1. Suppose that a figure beginning at (200,190) on the Hi-Res screen travels with a slope of one-half (moves up one and right two each time). Develop a table containing its X-Y coordinate pairs from its origin until it runs into the TOP of the screen. (In the course of moving it will contact the right side of the screen. At that time it should begin to move up one and left two with each movement.)
2. Use the table developed in Exercise 1 to animate the square from the shape table discussed in this chapter.
3. Use partial modification to cycle through ten digits on the Hi-Res screen.
4. Create a shape table containing a rectangle, its horizontal and vertical pre-shifts, its two diagonal (up one, over one) pre-shifts, and the pre-shifts required to move the rectangle up one, right two or up one, left two. (Use a graphics editor if you own one!)
5. Use each of the pre-shifts developed in Exercise 4 to move the rectangle.

14

Collision Course

Objectives

After reading Chapter 14 you should be able to:

- Detect a collision between objects on the screen using either the box method or the contact method.

By this time you should have your figures moving smoothly about the screen. In this chapter we will touch on a vital element of most every arcade game in existence: detecting a collision between two objects on the screen.

It is trivial for you to see when two objects meet; you simply look at the monitor! However, your Apple II does not have eyes, so it is in the same position as a blind person who uses a cane to probe for objects in his or her path. This analogy actually contains more truth than it has a right to.

Imagine the problem of dealing with just two objects on the screen. It is theoretically possible to keep a list of all the X-Y coordinates used in each shape, and then to check for collision by seeing if there is a coordinate pair which lies in both shapes. Suppose that one shape is plotted on [(1,1), (1,2),(1,3)], and the second uses coordinates [(1,2),(2,2),(3,2)]. In this case there must be a collision, since the point (1,2) is a part of both shapes. Sound simple?

Unfortunately, this method quickly becomes unworkable since, when animating, you are continually changing the points used in each shape, so you would have to continually change all of the lists. More important, the task of cross-checking each point gets quickly out of hand since even a trivial figure will contain perhaps ten points. Cross-checking two such

figures leads to 100 checks, and three figures may give you 1000! So much for that idea.

We are going to examine two methods which both use the blind-man's cane approach. We will fire a missile at a shape stolen from CROSSFIRE (by J. Sullivan), and as the missile moves, we will continually check its path. The shape table is given below, so take a few minutes now to enter and save it. From BASIC type:

```
CALL -151

300: 02 00 06 00 4A 00 4D 49
308: 49 69 18 DF DB DF DB 07
310: 48 49 69 4D 49 18 DF DB
318: DB DB 07 48 0D 6D 0D 6D
320: 0D D8 DB FB DF DB 48 09
328: 0D 0D 0D 4D 01 D8 DF FB
330: DF FB 08 4D 49 4D 49 05
338: 18 DF DB DB DB 07 08 4D
340: 49 49 49 05 D8 FB DB DB
348: DF 00 36 27 0D 36 00 00

BSAVE XFSHAPE, A$300,L$50
3 D0G
POKE 232,0:POKE 233,3
HGR:HCOLOR=3:SCALE=1:ROT=0
DRAW 1 AT 20,20
DRAW 2 AT 20 50
```

The first shape is our target. It should look a bit like a spider. The second shape is the missile, and it should look like this:

```
 X
XXX
XXX
```

When you get the table saved correctly, you are ready to animate. We will leave the spider alone and move the missile back and forth across the bottom of the screen using paddle 0. When the paddle button is pressed, we will move the missile up the screen until we bump into either the spider or the top of the screen. To effect that movement, we will plot and erase the missile, then move up one line and do it again. Before each movement of the missile, we will check the space it is moving into and see if there is something (the target) already there. Instead of checking to see if something hits the target, we will see if something runs into the missile!

We will develop the program in two stages. The following commands will get the basic movement started. You may omit any REM statements except 150 and 500 which are used as entry points for GOTO statements.

```
10 D$ = CHR$ (4)
20 PRINT D$ "BLOAD XFSHAPE"
30 M0 = 5:M = 5
80 REM
85 POKE 232,0: POKE 233,3
90 HGR : HCOLOR= 3: ROT= 0: SCALE= 1
100 DRAW 1 AT 140,50
110 XDRAW 2 AT M,150
120 REM
130 REM CHECK PADDLE
140 REM BUTTON
150 REM
160 IF PEEK (-16287) > 127 THEN
    GOSUB 500
170 REM
180 REM MOVE MISSILE
190 REM
200 IF PDL (0) < 90 THEN M = M-2:
    GOTO 220
210 IF PDL (0) > 150 THEN IF M < 277
    THEN M = M + 2
220 IF M < 1 THEN M = 1
230 IF M = M0 GOTO 250
240 XDRAW 2 AT M0,150: XDRAW 2 AT M,150:
    M0= M
250 GOTO 150
480 REM
490 REM COLLISION DETECT
500 REM
510 REM WE'LL PUT THIS IN SHORTLY!
520 PRINT CHR$(7):RETURN
```

Listing 14-1.

This listing will get the spider onto the screen and the missile moving under paddle control. When you press the paddle 0 button the computer will beep at you, which indicates that the (as yet non-existent) fire/collision-detect routine has been called.

M0 is the "old" X-coordinate of the missile, and M is the new one. The missile is moved by decrementing or incrementing M (lines 200 and 210), then XDRAWing to erase the old missile, and then XDRAWing again to plot the new one (line 240).

After you have the missile moving across the bottom of the screen, we will talk about the collision routine. Run the program as you have it so far, and press CTRL-C to recapture control while leaving the spider on the screen. Imagine that there is a little box drawn around the spider. In fact, put the box there by typing these four lines from BASIC.

```
HPLOT 137,38 TO 153,38
HPLOT TO 153,51
HPLOT TO 137,51
HPLOT TO 137,38
```

As we move the missile up the screen we will check to see if it moves inside that box, and if so—BOOM! To get that to happen, add these lines to your program:

```
4Ø REM
5Ø REM SET BOUNDARIES
6Ø REM
7Ø YMAX = 51:MINX = 137:MAXX = 153
51Ø PRINT CHR$ (7): REM BELL
52Ø C = Ø: REM COLLISION FLAG
53Ø FOR Y = 149 TO Ø STEP -1
54Ø IF Y < YMAX THEN IF M > MINX
    AND M < MAXX THEN C = 1: REM COLLISION!!
55Ø IF C THEN PRINT CHR$ (7); CHR$ (7);
    CHR$ (7): XDRAW 2 AT M,Y + 1: GOTO 62Ø
56Ø XDRAW 2 AT M,Y + 1: XDRAW 2 AT M,Y
57Ø NEXT Y
58Ø REM ERASE MISSILE @ TOP
59Ø Y = Y + 1
6ØØ XDRAW 2 AT M,Y
61Ø REM DRAW NEXT MISSILE
62Ø XDRAW 2 AT M,15Ø
63Ø RETURN
```

Listing 14-2.

Line 7Ø sets the boundaries for the square. The upper boundary is ignored in this example because the missiles always come from below.

Line 54Ø is the line that actually detects any collision. It first checks the Y-coordinate at the tip of the missile to see if it is high enough to possibly strike the box. If so, then the X position is checked to see if it is in the correct horizontal range. If the answer to all of these is "yes," then we have a collision and the flag (C) is set to 1 for later reference.

Line 55Ø is where you jump to get your nifty explosion routine, but since this is only an example, all we do is set off the bells and whistles. We then erase the missile from its last position and jump to where a new missile is drawn at the bottom—big deal!

In this case, since the target is stationary, it would be advantageous to check the X-coordinate before the Y. If the X value is within the range of the target, there is bound to be a collision. In that case you need to check only the Y-coordinate as the missile rises in order to tell you when the missile strikes the target. If the X is not within range, then the missile will

always miss the spider so you do not have to check either coordinate again, just let the missile make its way to the top of the screen. This sort of specialized adjustment can often streamline your animation.

The box method used in the example is fairly simple, but it is a little sloppy since it declares a collision based on some imaginary square instead of actual contact with the target. Further, we only checked for the center of the missile, and the center may actually miss the spider when one of the edges hits it. Unfair! If you have ever played a game which has sloppy collision detection (and there are a bunch of them) you know how frustrating that can be.

Checking across the full width of the missile is pretty easy, so we leave that to you. The problem of detecting an actual contact is more interesting. To accomplish that, you will have to take a PEEK at Hi-Res memory to see if the missile is about to move onto a dot which is already turned on. (In this example, the only dots turned on are in the target.) Dealing directly with memory from BASIC is always complex, and it gets worse because we need to look at individual bits. Type the following changes to the existing program.

```
DEL 40,70

5 GOSUB 1000: REM CALC Y ADDR.
515 Q% = M / 7:R% = M-7 * Q%:R% = R% + 1
536 V% = PEEK (Y%(Y) + Q%)
538 FOR I = 7 TO R% STEP -1
540 P% = 2 ^ I
542 IF V% > = P% THEN V% = V%-P%
544 NEXT I
546 IF V% > = 2 ^ I THEN C = 1
1000 REM
1010 REM CALCULATE Y
1020 REM COORDINATES
1030 REM
1040 DIM Y%(192)
1050 FOR I = 0 TO 184 STEP 8: READ SA%
1060 FOR J = 0 TO 7:Y%(I + J) = SA% +
     J * 1024
1070 NEXT J,I
1080 DATA 8192,8320,8448,8576,
     8704,8832,8960,9088
1090 DATA 8232,8360,8488,8616,
     8744,8872,9000,9128
1100 DATA 8272,8400,8528,8656,
     8784,8912,9040,9168
1110 RETURN
```

Lines 5 and 1000 to 1100 should be familiar from previous chapters; they

set up a table which assigns to each line on the screen (Ø—191) its starting address.

Line 515 divides the X-coordinate by 7 to determine which of the 4Ø bytes across the screen the missile is in (Q%), and which dot in that byte will be the point of the missile (R%). Let's suppose that the missile is fired at X-coordinate 45, so the tip is in byte number 6 and dot number 3 (counting the first dot in each byte as number Ø). As we move the missile up, we need to check the third dot in the byte above the missile before we move. As long as that dot is off, then there is nothing to hit. If that dot is on, then we are about to run into the target, and need to set the collision flag.

Isolating a particular dot is clumsy, but lines 338-546 do the job. Line 336 sets V% to the value of the byte you need to check. To understand how these lines operate, it is instructive (though tedious) to set V% to, say !ØØ11 11ØØ (6Ø), and step through those lines by hand. Here we go.

If the X-coordinate is 45, then line 515 calculates R% to be three (we are checking for dot number 3 in the byte), and then adds 1 to it so R% enters the loop at 538 with a value of four. Dot number 3 corresponds to the fourth bit from the right, yes the FOURTH bit, for they are numbered Ø, 1, 2, and 3.

P% is the value determined by two to the "I" power, and that just happens to correspond to each digit in a binary byte. Two to the eighth is the left-most bit (128), two to the seventh is the next bit over, and so on.

Each time the loop is executed, it effectively checks one of the bits in the byte and turns it off if it happens to be on. This continues until the bit indicated by R% is reached. Let's run through the example using Figure 14-1.

I	P%	V%
7	128	6Ø = !ØØ11 11ØØ
6	64	6Ø = !ØØ11 11ØØ
5	32	28 = !ØØØ1 11ØØ
4	16	12 = !ØØØØ 11ØØ
3	-	12 = !ØØØØ 11ØØ

Figure 14-1.

The first time through the loop, I = 7, so P% = 128. Line 542 finds the comparison (IF V% > = P%) false, so it does not subtract anything from V%. The second time through, P% = 64, still greater than V%, so again nothing happens. But on the third iteration, I = 5, P% = 32, so P% is subtracted from V% to reset V% at !ØØØ1 11ØØ. As promised, the bit was

turned off. The next time through, when I = 4, the next bit is also turned off. After that the loop is halted since I has reached the value of R%.

Therefore, at the end of the loop, V% equals 12, which in binary is !0000 1100. We were trying to check the fourth bit from the right, and we do that now with line 546. The left-over value of I is three, so V% is compared against two to the third, or eight. The only way for V% to be greater than or equal to eight is for the fourth bit to be on, so the collision flag is set. Whew!

I	P%	V%
7	128	52 = !0011 0100
6	64	52 = !0011 0100
5	32	20 = !0001 0100
4	16	4 = !0000 0100
3	-	4 = !0000 0100

Figure 14-2.

Figure 14–2 shows the same procedure performed when V% = 52 where the fourth bit is off. There should not be a collision indicated. If you follow the loop through, you will see that the comparison at line 546 fails, so the flag is not set. We told you that it would be rather awkward to check single bits!

One weakness of this routine (as in the previous one) is that it only checks the center of the missile, so it is possible to contact the target with the left or right sides and still not detect a collision. Two trivial modifications to the routine will cause it to check across the entire width of the missile, but again we leave them for you to discover!

When you run this version of the program, you will notice that a collision is detected only when the missile actually contacts the target. You will also notice that the animation is slower due to the increased processing involved in this scheme—you can't have everything.

With a little effort, the BASIC code can be streamlined to enhance the execution, but the real gain is made by writing this collision detect in machine code, something we will not cover here.

The contact collision detect is further complicated if you have other objects on the screen, such as clouds, where the missile is required to ignore the contact. There are two commonly used ways to handle this.

In THRESHOLD, a table is kept of the current position for each target. When any collision is flagged, that table is scanned to see if there is a target close. If so it is destroyed, and if not the collision is ignored. One interesting side effect is that when two THRESHOLD targets are close

together, the scanning routine will sometimes decide that the wrong one has been shot.

The other method of dealing with extraneous objects is to use both Hi-Res screens. One screen is displayed with all the missiles, targets and other junk on it, while the other has only the missiles and targets on it, and is not displayed. The collision detect routine can then check the second screen and not even worry about the extra figures! In some applications the extra time needed to process two screens is offset by not having to decide what you ran into.

In Summary....

That about wraps it up. Hopefully this book has shed some light on ways to make Apple graphics do what you want them to, and how many of the games achieve their effects.

So now you have it—the tricks of the animation trade. The ideas presented here are the most common and effective techniques currently used with the APPLE II. Programming personal computers is a creative and fast-paced sport. It is limited only by the programmer's imagination, so we can expect that before too long there will be new, improved methods—that's (programming) life! However, the animation techniques you have learned here will still be useful, and the skills gained will help you to understand and implement the new approaches.

Vocabulary

Box method

Collision flag

Contact method

Exercises

1. Use the table which contains the apple shape (from Chapter 10) to animate two apples on the screen, each under control of one of the paddles. (If you do not have paddles, control the apple's movement with the keyboard.) Write a collision detect routine using either the box or contact method to determine when the two shapes collide.
2. Use the program from Chapter 13 (Listing 13-2), which moves the byte-move line across the screen, and draw the spider shape in its path. Write a routine to signal when they contact.
3. Write a computer game and make a million bucks.

Appendix 1 Decimal, Hex, and Binary

We are assuming that you have read the discussion of binary and hexadecimal in Chapter 2, and that you have a reasonable grasp of that material. In this section we will delve deeper into the structure of the three systems of numeration in order to give you the background needed to understand the techniques needed to transform the numbers used by your Apple II. For instance, why is the decimal form of an address found from the hex form by taking 256 times the decimal value of the hi-byte plus the decimal value of the lo-byte? And how can anybody say that 38400 is equivalent to -27136??! All of this will be explained ... and more. But first we are going to acquire some background which sounds, and is, distressingly like the New Math.

Positions, Everyone

When you were just a young sprout, your teachers probably explained to you the significance of a digit's position within a number. Five does not always mean five; the five in 56 indicates fifty, but more precisely, five tens. In 567, the 5 represents five hundreds, and in 5,678 it represents five thousands. The position of the digit within the number is significant. From right to left, you have the unit's place, the ten's place, hundred's place, and so forth. So 5,678 can be written as

$$(5 \times 10^3) + (6 \times 10^2) + (7 \times 10^1) + (8 \times 10^0)$$

which is

$$(5 \times 1000) + (6 \times 100) + (7 \times 10) + (8 \times 1)$$

Many of us unfortunates spent hours of our young lives writing numbers that way, called "expanded form," for no apparent reason.

There is little or no use in writing decimal numbers in that form, but your fourth grade teacher (Miss Whitherstare, or whoever) did you an inadvertent favor by making you learn it, for expanded form is one way to convert numbers from other outlandish bases, like binary and hex, to decimal. To convert a long hex number, like $89AB, to decimal, write it as

$$(8 \times 16^3) + (9 \times 16^2) + (A \times 16^1) + (B \times 16^0)$$

Then convert each hex digit to decimal which gives you

(8 x 4Ø96) + (9 x 256) + (1Ø x 16) + (11 x 1) = 35243.

There is a variation on this method which is handy when you are proficient with hex bytes. When dealing with bytes, recall from Chapter 2 we mentioned that two hex digits could be used to represent one base 256 digit. Then the address given by $FF3A can be calculated as

($FF x 256^1) + ($3A x 256^0)
= (255 x 256) + (58 x 1)
= 35243

To use this idea efficiently, you must know or be able to quickly calculate the value of numbers like $3A—a skill that comes with practice.

The expanded form method also works when converting binary to decimal, but since binary is base two, you get two to all those powers instead of sixteen. From right to left in binary, the first position is the ones (2^0) place, then the two's (2^1) place, the four's (2^2) place, the eight's (2^3) place, and so on. Therefore, you may write !11Ø1Ø1 as

(1 x 2^5) + (1 x 2^4) + (Ø x 2^3) + (1 x 2^2)
+ (Ø x 2^1) + (1 x 2^0)

Or even simpler, since it is either one or zero times the place value

32 + 16 + 4 + 1
= 53

This was calculated by adding the place values with ones in them, and ignoring the places containing zeros.

Wonderful, but you already knew how to convert from binary and hex to decimal. These ideas will help you to become more familiar with what you are doing, and make you faster and more flexible in your conversions. Converting to decimal is fairly straightforward, but unfortunately the same is not true of converting from decimal to binary or hex.

Decimal to Binary

When converting to binary, you can start by finding the highest power of two which will go into the decimal number—if you can get your hands on a calculator, now would be a good time to do so. Let's take 421 as an example to convert: 2^9 (512) is too large to divide by 421, but 2^8 = 256 (a number to know in computing), and 256 will divide 421, albeit with a remainder of 165 (421-256), so we have a 1 in the 2^8 place which will be on the left end of our binary number. The next lowest power of two is 128, which divides 165 leaving a remainder of 37. So far our binary number is 11xxxxxxx. Dropping to the next power of two (64) we see that it will not divide the present remainder (37) so we put a zero in the next place in our

number (11Øxxxxxx). 32 will divide 37 to yield a 1 in the next place (11Ø1xxxxx) and a new remainder of 5. 16 and 8 both fail to divide the 5, so we place two more zeros (11Ø1ØØxxx). 4 divides the 5, 2 fails to divide the remainder (1), and we are left with a 1. So the last three digits are 1Ø1, to give us the binary equivalent of 421 as !11Ø1ØØ1Ø1. The algorithm is tedious, but simple—a good candidate for a computer program!

REMEMBER: When converting decimal to binary, divide by the largest possible power of two to form the quotient and remainder (no decimal points!). Take the remainder of the first division and attempt to divide it by the next lower power of two, and so on, attempting to divide each remainder by the next power of two until you reach a division by 1. Any successful division gives you a 1 in the corresponding place, while an unsuccessful division yields a Ø in that place.

Decimal to Hex

Converting decimal to hex follows the same general outline as decimal to binary, but it is complicated by having sixteen possible values for each digit instead of two as in binary. When you were dividing by powers of two, each power of two went into your number either once, to give you a 1, or not at all, which gives you a Ø. If your power divided the number more than once (two or more), then you knew that you should have been dividing by a higher power of two.

The unfortunate who converts decimal to hex finds himself or herself dividing by powers of sixteen (1, 16, 256, 4Ø96), and a power of sixteen may divide the number anywhere from zero to fifteen times! For example, let's convert 1Ø19 to hex.

256 is the largest power of sixteen which will divide 1Ø19, and when you perform that division on your calculator, the answer is 3.98Ø4 and so on, so 256 goes into 1Ø19 three times with something remaining. That means you put a 3 in the 256's place (What 256's place?!). Since this is the first division, the 3 is on the left end of the hex number, and now you need to know the remainder of the division. To find that (since your calculator is grossly inaccurate and truncates after ten or fifteen decimal places), multiply three by 256 and subtract the result from 1Ø19 (1Ø19—3x256), to get 251.

Now divide 251 by the next power of 16 (which happens to be 16 itself), and your calculator will display a fifteen, a decimal point, and garbage. The fifteen indicates that you have an "F" (the hex equivalent of 15) in the next place of your hex number, which so far looks like $3F, and the decimal point garbage means that you must calculate a new remainder by multiplying sixteen times "F" (15), and subtracting from 251 (251—15 x 16) to get 11. Therefore, the last digit of the hex number is "B" (the hex equivalent of eleven, silly!), so the entire number is $3FB. Still a little hazy,

you say? It takes some practice before you will be comfortable with hex, unless you were born with sixteen fingers!

Let's convert 41141 to hex. 4Ø96 (16^3) is the largest power to divide 41141, and it does so ten times with a remainder, so the left-most digit will be $A (hex for 1Ø). Calculating the remainder (41141—1Ø x 4Ø96) gives you 181. The next power of sixteen (16^2 or 256) will not divide 181 at all, so put a zero in the place following the "A" and get $AØ. Dividing 181 by 16 gives 11 with a remainder of 5, so the last two digits are "B" and 5, and the entire number is $AØB5. TA DA!!

In a Nutshell:

To convert a decimal number to hex, divide the decimal number by successively smaller powers of sixteen, each time using the quotient as the hex digit, and the remainder as the decimal number for the next division.

There is an alternate method which uses hex to help convert large ugly decimal numbers to large ugly binary numbers and vice versa. Take 65385 for a sample. Start by converting to hex as we showed you earlier:

65385 = (15 x 4Ø96) + (15 x 256) + (6 x 16) + (9 x 1)
= $FF66

Do not take our word for it—try it yourself! The next step is to convert each of the four hexadecimal digits to binary using the chart we gave you in Chapter 2: $F = !1111, $6 = Ø11Ø, and $9 = !1ØØ1. Armed with this information you can write the entire number in binary: 65385 = $FF69 = !1111 1111 Ø11Ø 1ØØ1. Using hex as an intermediate step saves you from having to calculate the highest power of two which goes into 65385. (Turns out to be 2^{15} or 32768!)

To convert a large binary number (say !1ØØ1 Ø11Ø ØØØØ ØØØØ) to decimal we use the same scheme in reverse. That binary number is $96ØØ in hex, which in turn is (9 x 4Ø96) + (6 x 256) + (Ø x 16) + (Ø x 1) = 384ØØ.

Adding and Subtracting Hex

When working with the computer's memory, you will at some point or other be forced to add and/or subtract addresses. An example is given in the discussion of shape tables where it is suggested that you might load multiple tables into memory consecutively, and then place the HIMEM boundary below them in order to protect the tables from the machinations of APPLESOFT. Do not panic if you do not know about shape tables yet. For this application just think of a shape table as a bunch of bytes that you want to load into memory.

Let's suppose that you have two tables to load, one $876 bytes long and

the other $5A7 bytes. You will first add the two lengths together, and then subtract the total from the HIMEM value—$96ØØ if you have 48K and DOS.

There are several methods used to perform the calculations. The way most of us began was to convert all the numbers to decimal, perform the addition and/or subtraction, and then convert to hex again—a safe, though time-consuming approach. The other extreme is to do the entire operation in hex, and very few people can actually carry that off. The technique we will demonstrate leaves the numbers in hex, but you mentally convert each of the digits as you go. Write the two numbers to be added so that one is atop the other, as below.

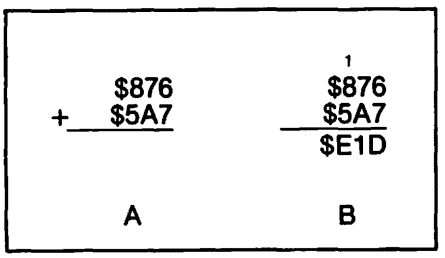


Figure A1-1.

You begin, as usual, from the right and add 6 and 7 to get thirteen. Miss Whitherstare told you that thirteen was "three carry one", but (aha!!) you are working in hex, so thirteen is written as "D" and there is no carry. Add the next two digits, 7 and "A" to get seventeen. Seventeen is one more than the magic number for hex (sixteen), so you write down the "1" and carry the sixteen over to the next column as a one (See Figure A1-1B). If that sounds strange, remember how you handle 7 + 4 in decimal: the sum is eleven (ten plus one), so you write the one down and carry the ten over as a one in the next column!

In base ten you carry tens to the next column, and in base sixteen (hex) you carry sixteens.

We are down to adding the last column, in which you add the 8, the 5, and the 1 from the carry, and get fourteen which is $E. This gives you $E1D as the combined length of the two tables. You are doing very well so far.

Do not get too confident though, you still need to subtract that amount from HIMEM ($96ØØ) to find the new value to use.

The first subtraction pits Ø against "D", and you have the age-old problem of borrowing, but in hex you borrow sixteens instead of tens. If you attended public elementary school after 1959 you are probably used to borrowing across the top, but if you do it differently, bear with us—the

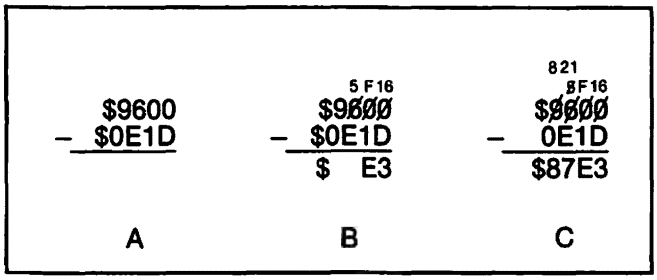


Figure A1-2.

idea is the same. You borrow one from the 6 and turn it into sixteen in the next column to the right, while the 6 becomes a 5. You then borrow one from the sixteen, which becomes a fifteen ($F), and the borrowed one becomes sixteen in the right-most column—Arrgh! (See Figure A1-2B.)

Now that the preliminaries are over, we begin the subtraction in earnest with the right hand column: sixteen subtract "D" (thirteen) is three. The next column shows F (fifteen) subtract 1, which gives E (fourteen) as that result. The next column again requires a borrow, so borrow one from the 9, making it 8, and add sixteen to the five to get 21 (decimal!), as in Figure A1-3C. Now you can calculate twenty-one less "E" (fourteen) to get 7, the eight carries down, and so the final result is $87E3. If you got a different answer, stay after class and clean the erasers!

Let's try subtracting $4D from $3A7 for practice. Again the numbers are lined up, and you quickly see that a borrow is going to be needed (drat!).

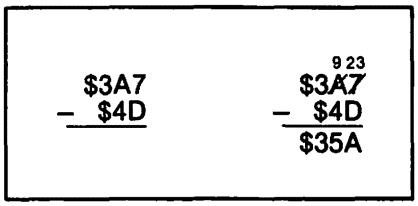


Figure A1-3.

Borrowing hex one from the A leaves it a nine, and carries a decimal sixteen over to the seven to get decimal twenty-three. Twenty-three minus D (13) is A (10). Then you take four from nine to get five, and carry the three down. The final answer is $35A.

Adding and Subtracting Binary

Having to perform binary arithmetic is a curse that often befalls programmers, even though the machine is supposed to do all the work. When faced with this proposition, you should keep in mind that zero and one are

the only available digits, and that a binary two is written as 1Ø, and binary three is 11. Why are those important? We will demonstrate by adding !ØØ11 Ø11Ø to !ØØØ1 11Ø1.

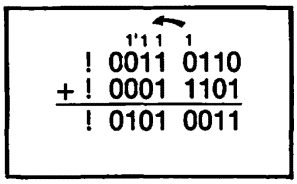


Figure A1-4.

Beginning at the right side, the first column is Ø + 1 which gives you 1. You should also get a one in the second column without too much trouble—so far, so good. The third column gets more complex as you have 1 + 1. Any normal second-grader can tell you that 1 + 1 = 2, but programmers should make no pretense at normalcy. As we mentioned above, two is written as 1Ø (one-zero, not ten!), which puts the zero in the answer and carries the one to the next column. Including the carry, that next column now requires you to add 1 + Ø + 1, which again gives 1Ø (write down the zero and carry the one). The one is carried over to the fifth column where you now have 1 + 1 + 1 which yields three (11). For that, you must write down a one and carry a one to the next column which then looks like 1 + 1 + Ø, and requires you to write down a zero and carry a one (1Ø). Then the seventh column is 1 + Ø + Ø which is a one (and no carry) and the last column is Ø + Ø which is zero. That gives us a result of !Ø1Ø1 ØØ11. WOW!

Figure A1-5 shows another example of adding binary, this time with a carry off the end.

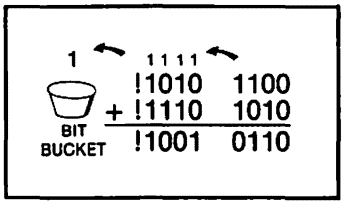


Figure A1-5.

The first column (right-most) gives zero, and both of the next two columns give ones. See how simple life is without a carry? The fourth column gives zero, carry one, and the fifth then gives one with no carry. Things warm up a little with the sixth column which gives zero, and carries one to the seventh. That column then also gives zero and carries on to the eighth column. This makes the eighth addition 1 + 1 + 1 which inevitably gives 11—write down one and carry one. Since a byte is only eight bits long, the bit you try to carry to the ninth column falls into the bit bucket

and is ignored. For any of you interested in machine-level programming, when a bit is carried off the end of an addition, the processor sets a signal bit, called the carry flag, within the status register. That signals (to anybody who knows where to look) that the latest operation had to dump a carry into the bucket.

Subtraction

Figure A1-6 shows an example of binary subtraction.

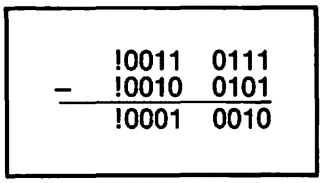


Figure A1-6.

Again you work from right to left, and you should have no problem arriving at the result. This sample was carefully constructed so that you could subtract without having to fiddle around with borrowing, not because of cowardice, but because you never really have to worry about borrowing in binary. The fact is, your computer never subtracts, so why should you? Instead of subtracting a number, the arithmetic unit of your machine always ADDS a thing called the "two's complement."

Two's Complement

Forming the two's complement of a binary number takes two steps: first change all the ones to zeros, and all the zeros to ones, and second, add one to the result. To form the two's complement of !0010 0101, you first turn it into !1101 1010 and then add 1 to get !1101 1011. Now we can repeat the example from Figure A1-6 using addition, but instead of subtracting !0010 0101, you will add its two's complement, !1101 1011.

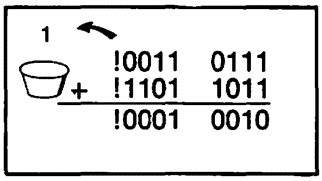


Figure A1-7.

The addition is shown in Figure A1-7, and is fairly straightforward. The answer comes out the same as before, except for the carry from the last

column which falls into the bit bucket that is ignored anyway. This is why the two's complement can be used as if it were a negative number, when it really is not.

If you add any number to its corresponding negative value (five plus negative five) you will always get zero. The same thing happens when you add any binary number to its two's complement ... almost. Try adding

```
!ØØ1Ø Ø1Ø1
```

and its complement:

```
!11Ø1 1Ø11
```

You will get all zeros, except for the final carry which falls into the bucket. So we always expect that carry, and ignore it when it occurs.

Figure A1-8 shows another sample of subtraction using two's complement.

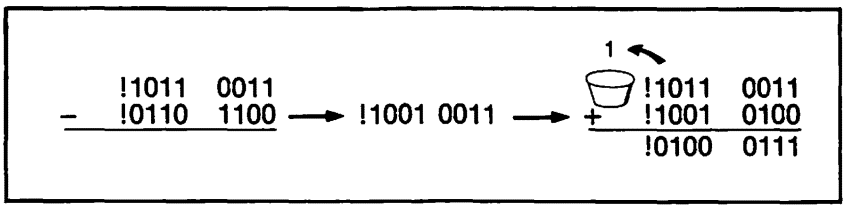


Figure A1-8.

Notice that it is always the number being subtracted which is changed to its two's complement form. !Ø11Ø 11ØØ is changed to !1ØØ1 ØØ11, and then a 1 is added to get !1ØØ1 Ø1ØØ. This number is then added to !1Ø11 ØØ11 to get !Ø1ØØ Ø111, and as usual we ignore the bit in the bucket.

Negative Addresses

The two's complement essentially changes a number to its negative equivalent, so instead of subtracting a number you end up adding its negative. In a decimal setting you can think of it as changing the subtraction 9—3 into the addition 9 + (-3) and the result is six in either case.

The same idea lets some addresses be written in negative form for convenience. When you wish to enter Monitor from BASIC you CALL the address -151. The true address for the Monitor entry point is 65385, and -151 is the two's complement of that address. If you convert 151 to its two-byte binary form (!ØØØØ ØØØØ 1ØØ1 Ø111) and take the two's complement (to account for the negative) you get !1111 1111 Ø11Ø 1ØØ1. The not-so-quick conversion to decimal results in the expected value of 65385. You must admit that -151 is much easier to remember that 65385.

Another reason for using the two's complement, or negative, form for

some of the addresses is that the original language for the Apple II, Integer BASIC, was incapable of handling a number greater than 32767. This meant that in order to CALL the 65385 address, you had to refer to its two's complement, -151.

You can convert any address to its negative equivalent by either subtracting 65536 from the address (the easy way) or by forming its two's complement (the hard way).

At this point you have probably learned more than you ever wanted to know about the hex and binary systems, but you have really just scratched the surface. There are several interesting ties between the topics we covered here and the fundamental algorithms used by a computer. For example, a computer actually figures the sum of two numbers by comparing corresponding bits and using the logical operations "AND" and "EOR" (described in the text). Pursuing these topics would take us beyond the scope of this text, but that information may be found in a book on Boolean logic.

Appendix 2
Character Codes: ASCII vs APPLE

Table A2-1. Apple Screen Characters.

	Inverse				Flashing				Normal (Control)							Normal (Lowercase)
Decimal	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
Hex	$00	$10	$20	$30	$40	$50	$60	$70	$80	$90	$A0	$B0	$C0	$D0	$E0	$F0
0 50	@	P		0	@	P		0	@	P		0	@	P		0
1 $1	A	Q	!	1	A	Q	!	1	A	Q	!	1	A	Q	!	1
2 $2	B	R	"	2	B	R	"	2	B	R	"	2	B	R	"	2
3 $3	C	S	#	3	C	S	#	3	C	S	#	3	C	S	#	3
4 $4	D	T	$	4	D	T	$	4	D	T	$	4	D	T	$	4
5 $5	E	U	%	5	E	U	%	5	E	U	%	5	E	U	%	5
6 $6	F	V	&	6	F	V	&	6	F	V	&	6	F	V	&	6
7 $7	G	W	'	7	G	W	'	7	G	W	'	7	G	W	'	7
8 $8	H	X	(	8	H	X	(	8	H	X	(	8	H	X	(	8
9 $9	I	Y	)	9	I	Y	)	9	I	Y	)	9	I	Y	)	9
10 $A	J	Z	*	:	J	Z	*	:	J	Z	*	:	J	Z	*	:
11 $B	K	[	+	;	K	[	+	;	K	[	+	;	K	[	+	;
12 $C	L	\	,	<	L	\	,	<	L	\	,	<	L	\	,	<
13 $D	M	]	-	=	M	]	-	=	M	]	-	=	M	]	-	=
14 $E	N	^	.	>	N	^	.	>	N	^	.	>	N	^	.	>
15 $F	O	_	/	?	O	_	/	?	O	_	/	?	O	_	/	?

(Used with permission of Apple Computer Corporation)

Table A2-2. ASCII Character Codes.

DEC is ASCII decimal code
HEX is ASCII hexadecimal code
CHAR is ASCII character name
n/a = not accessible directly from the APPLE II keyboard

DEC	HEX	CHAR	WHAT TO TYPE	DEC	HEX	CHAR	WHAT TO TYPE
0	00	NULL	ctrl @	48	30	0	0
1	01	SOH	ctrl A	49	31	1	1
2	02	STX	ctrl B	50	32	2	2
3	03	ETX	ctrl C	51	33	3	3
4	04	ET	ctrl D	52	34	4	4
5	05	ENQ	ctrl E	53	35	5	5
6	06	ACK	ctrl F	54	36	6	6
7	07	BEL	ctrl G	55	37	7	7
8	08	BS	ctrl H *or* ←	56	38	8	8
9	09	HT	ctrl I	57	39	9	9
10	0A	LF	ctrl J	58	3A	:	:
11	0B	VT	ctrl K	59	3B	;	;
12	0C	FF	ctrl L	60	3C	<	<
13	0D	CR	ctrl M *or* RETURN	61	3D	=	=
14	0E	SO	ctrl N	62	3E	>	>
15	0F	SI	ctrl O	63	3F	?	?
16	10	DLE	ctrl P	64	40	@	@
17	11	DC1	ctrl Q	65	41	A	A
18	12	DC2	ctrl R	66	42	B	B
19	13	DC3	ctrl S	67	43	C	C
20	14	DC4	ctrl T	68	44	D	D
21	15	NAK	ctrl U *or* →	69	45	E	E
22	16	SYN	ctrl V	70	46	F	F
23	17	ETB	ctrl W	71	47	G	G
24	18	CAN	ctrl X	72	48	H	H
25	19	EM	ctrl Y	73	49	I	I
26	1A	SUB	ctrl Z	74	4A	J	J
27	1B	ESCAPE	ESC	75	4B	K	K
28	1C	FS	n/a	76	4C	L	L
29	1D	GS	ctrl shift-M	77	4D	M	M
30	1E	RS	ctrl ^	78	4E	N	N
31	1F	US	n/a	79	4F	O	O
32	20	SPACE	space	80	50	P	P
33	21	!	!	81	51	Q	Q
34	22	"	"	82	52	R	R
35	23	#	#	83	53	S	S
36	24	$	$	84	54	T	T
37	25	%	%	85	55	U	U
38	26	&	&	86	56	V	V
39	27	'	'	87	57	W	W
40	28	(	(	88	58	X	X
41	29	)	)	89	59	Y	Y
42	2A	*	*	90	5A	Z	Z
43	2B	+	+	91	5B	[	n/a
44	2C	,	,	92	5C	\	n/a
45	2D	-	-	93	5D	]	] (shift-M)
46	2E	.	.	94	5E	^	^
47	2F	/	/	95	5F	–	n/a

(Used with permission of Apple Computer Corporation)

Appendix 3
Memory Maps

Memory Map of a 48K Apple II

Function	Address
APPLE MONITOR	$F800-$FFFF
APPLESOFT	$E000-$F7FF
RESERVED	$D000-$DFFF
I/O DECODE	$C000-$CFFF
DOS	$9600-$BFFF
UNUSED	$6000-$95FF
HI-RES PAGE 2	$4000-$5FFF
HI-RES PAGE 1	$2000-$3FFF
UNUSED	$C00-$1FFF
TEXT/LO-RES PAGE 2	$800-$BFF
TEXT/LO-RES PAGE 1	$400-$7FF
DOS VECTORS	$3C0-$3FF
UNUSED	$300-$3BF
TEXT INPUT BUFFER	$200-$2FF
6502 STACK	$100-$1FF
ZERO PAGE	$0-$FF

Figure A3-1.

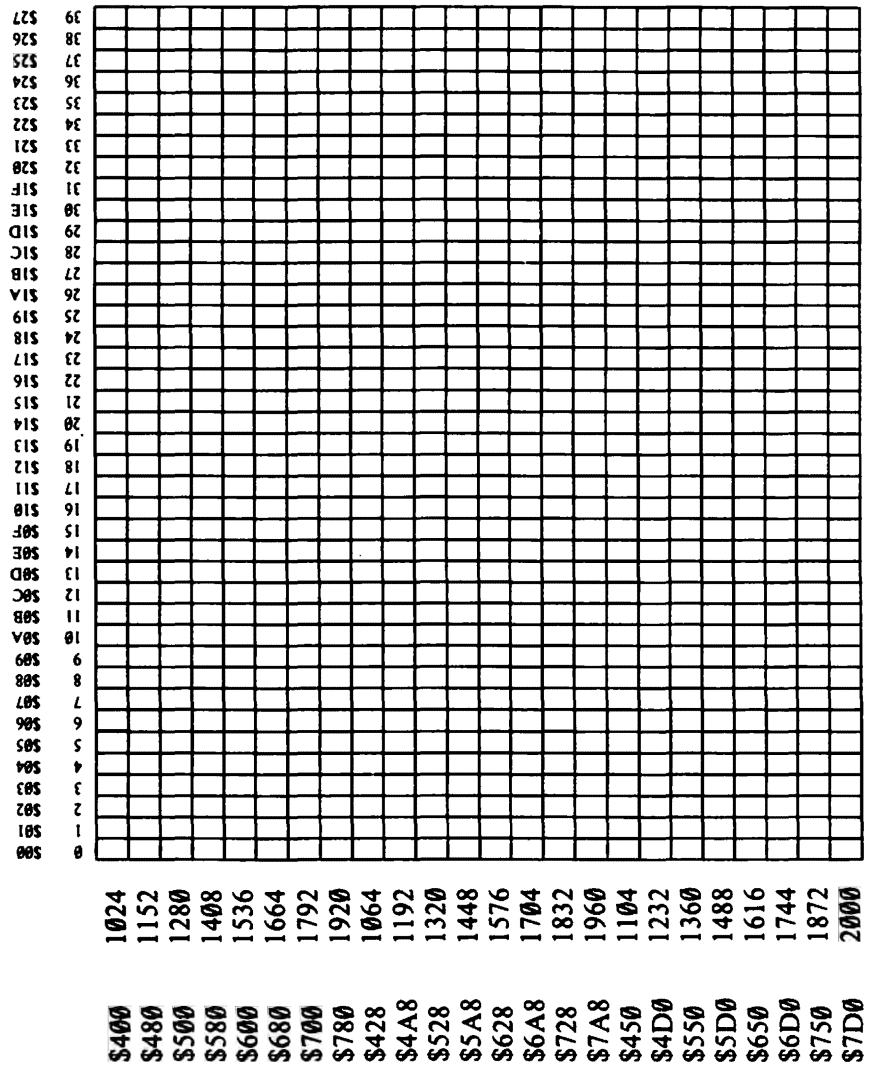


Figure A3-2.

(Used with permission of Apple Computer Corporation)

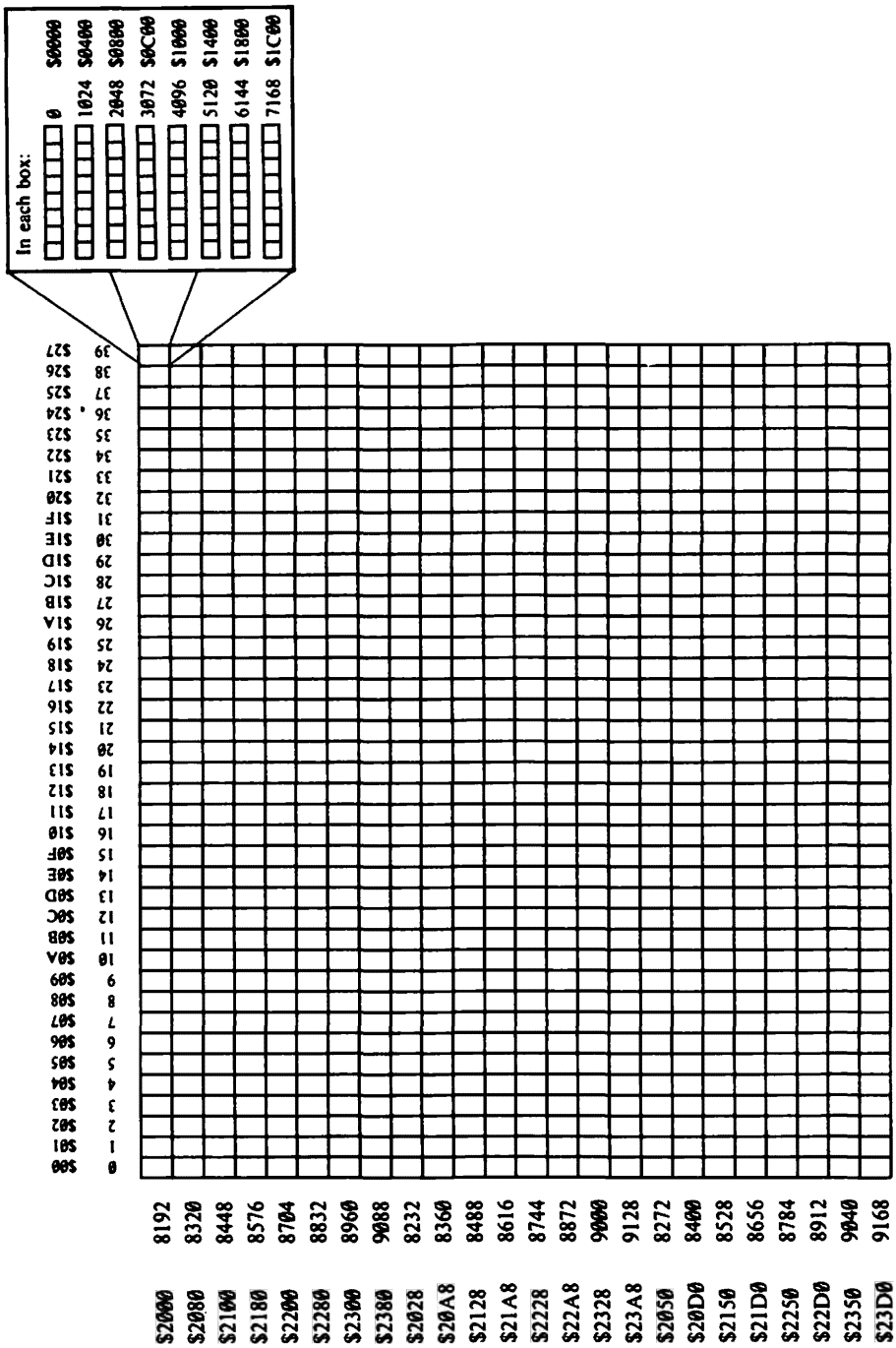


Figure A3-3.

(Used with permission of Apple Computer Corporation)

To obtain the address for any byte, add the addresses for that byte's box row, box column, and position in box.

Glossary

ADDRESS. A particular number that corresponds to each byte of memory.

ANIMATE. To make an object seem alive, to cause it to move about.

ASCII. American Standard Code for Information Interchange; all the numbers and letters used on a computer are assigned numeric values. See page 138 of the *APPLESOFT Reference Manual.*

BINARY. A system of numeration having only two characters, Ø and 1. Any number may be represented using multiple digits. For example, decimal 3 is binary 11 and decimal 5 is binary 1Ø1. Binary files are files stored in binary form.

BIT. The fundamental unit of memory in a computer. A bit can be in one of two states—a high or low voltage. It is normally represented as either 1 or Ø.

BLOAD. The command to load a binary file from disk or tape.

BOX METHOD. A method of detecting collisions where you consider any shape entering a region around the target defined by a range of X-coordinates, and a range of Y-coordinates, to collide with a target.

BSAVE. The command used to save a binary file on disk or tape.

BUFFER. A temporary storage area.

BYTE. A unit of information in memory made up of eight bits. For example, in binary, 1101 0111; in hexadecimal B7. A byte may contain any value between Ø and 255 decimal, inclusive.

BYTE-MOVE. An animation technique in which the digitized shape is stored in memory and then moved into Hi-Res memory for display.

CALL. A BASIC statement similar to GOSUB, except it executes a machine-level routine as opposed to a BASIC routine.

CLASHING. The result of trying to plot colors from different color groups together in the same byte of memory. The results of clashing are unpredictable colors.

COLLISION. In animation, when one shape runs into another.

COLLISION FLAG. A value which is set by your routine to signal that a collision has occurred.

COLOR BIT. The high-order (left-most) bit in a byte of the Hi-Res screen memory. The color bit determines to which of the two color groups the byte belongs to.

COLOR GROUP. Bytes displayed on the Hi-Res screen belong to one of two color groups. Bytes in group 1 may display dots which are black, white, green, or violet; bytes in group 2 may display black, white, blue, or orange.

COLUMN ADDRESS. The portion of the address of a byte on the graphics screen which must be determined by that byte's position across the screen.

CONTACT METHOD. A method for detecting collisions only when one shape actually contacts another, as opposed to the Box Method.

COORDINATES, X and Y. The X-coordinate indicates a horizontal position, and the Y-coordinate indicates a vertical position.

DECIMAL. The normal system of numeration which uses ten digits, Ø through 9.

DIGITIZE. To create a series of data values for a figure or other non-numeric item.

DITHERING. A technique which places several of the the six elementary Apple Hi-Res colors in a pattern so it produces non-standard colors.

DOS. An acronym for *disk operating system,* the controlling programs that make your disks behave properly (usually).

DRAW. An APPLESOFT command which draws a shape on the Hi-Res screen.

EDITOR. A program to simplify creating, altering, and saving either text or graphics.

EXCLUSIVE OR (EOR). A Boolean (binary) operation which compares two bits and results in an ON state if either of the compared bits are ON but not if both are ON.

EXEC. A DOS command which causes the computer to process the contents of a text file in the same way it usually handles keyboard input.

FLICKER. Flash on and off quickly. Flicker is the bane of animation programmers.

FULL-SCREEN GRAPHICS. The graphics screens without four lines of text at the bottom.

HEX or **HEXADECIMAL.** Referring to hexadecimal; a system of numeration using 16 different characters for digits: Ø through 9 and A through F. Hex numbers are conventionally preceded by a "$" to differentiate them from other types of numbers. $A is decimal 1Ø, $B is decimal 11, and so forth.

HIMEM. The value which represents the top of user-available memory. The APPLESOFT variables use HIMEM as a starting point.

HI-RES. High Resolution graphics. In Hi-Res graphics, all objects displayed are made up of a number of small dots on the screen.

HI-RES PAGES. The area in memory which may be displayed as pictures on the video output.

HI-RES ROUTINE. A program designed to manipulate the contents of the Hi-Res pages of the Apple memory.

HI-ORDER BYTE. In an address, the most significant (left-most) byte. For example, in the hex address $A3ØF, A3 is the hi-order byte (hi-byte).

HLIN. A BASIC command used to draw a horizontal line in Low-Res.

LO-BYTE/HI-BYTE. The form Apple uses for hex addresses; the low-order byte comes before the hi-order byte.

LOW-RES. A graphics mode of the Apple II where the video output is a number of small colored blocks.

LOW-ORDER BYTE. In an address, the least significant (right-most) byte. For example, in the hex address $A3ØF, ØF is the low-order byte (lo-byte).

MEMORY. Data storage area within the computer.

MEMORY MAP. A diagram of the computer's memory.

MIXED SCREEN GRAPHICS. The graphics screen with four lines of text at the bottom.

MONITOR. The ROM resident machine-level language in your Apple. If written without the capital "M," the video display.

MOVE. A command available from Monitor to move a block of memory.

NIBBLE/NYBBLE. Either of the first or last four bits in a byte—half of a byte.

OFFSET. A value added to the base address to locate a byte with respect to that base address.

OPTIMIST. A programmer who codes in pen.

PAGE. A unit of memory. 256 bytes of memory make up one page. When used in the context of graphics display, a page refers not to 256 bytes, but to the area of memory which is to be displayed on the video screen.

PARTIAL MODIFICATION. An animation technique where only the portions of the figure which change are redrawn. The rest are left alone.

PEEK. A BASIC command which looks at a specified address in memory and returns the value that it finds there.

PIXEL. The smallest unit of video display controllable by the computer.

PLOT. To put something on the screen.

POINT OF ORIGIN. The point within a shape where the defining vectors begin.

POKE. A BASIC command which puts a specified value into a given location.

POSITION ADDRESS. The portion of the address for a byte on the Hi-Res screen which is given by that byte's position within its box. See the Hi-Res memory map.

PRE-CALCULATION. A method for reducing flicker. All the calculations needed for the animation are done before the figure begins to move. The results are stored in a table for reference.

PRE-SHIFT. (1) The result of DRAWing the shape once, shifting the shape in any direction, and then XDRAWing over the original. (2) A way of animating a shape by XDRAWing the necessary changes on top of it.

PRIMARY PAGE. The first of two pages of either graphics or text.

PROMPT CHARACTER. The character the Apple prints on your screen to remind you which language you are speaking to. Either APPLESOFT (]), Integer BASIC (>), or Monitor (*).

QUOTIENT. The result of a division.

REMAINDER. The result of subtracting the product of the divisor and the integer quotient from the number originally divided. Loosely (and much more understandably), what is left over after an integer division.

RAM. An acronym for *random access memory*. RAM may be altered by the program.

ROM. An acronym for *read only memory*. ROM cannot be changed by software—you cannot write to it, you may only read it.

ROM RESIDENT. Something stored in ROM; therefore, it is permanent.

ROT. BASIC's Hi-Res shape rotation command. A shape rotates about its point of origin.

ROUTINE. A program designed to accomplish a particular task.

ROW ADDRESS. A portion of a graphics byte's address given by the row which the byte's box occupies on the memory map.

SCALE. A BASIC command used to enlarge shapes.

SECONDARY PAGE. The second of two pages of either graphics or text.

SECTOR. A block of storage on a disk. One sector holds 256 bytes of data. DOS allows for 496 sectors to be available to the user.

SEPARATION. Any one of the seven versions of a figure required to move a byte-move figure horizontally. Each separation is shifted across one screen dot from the previous one.

SHAPE. SHAPE is an Apple graphics construct. Any figure on the screen can be digitized so that it will operate with the APPLESOFT DRAW, XDRAW, ROT, and SCALE commands. Digitizing the shape can be a complex process, but a Hi-Res editor takes care of all the grubby details so that creating shapes is a quick, simple task.

SHAPE TABLE. A number of shapes strung together. Again, a Hi-Res editor makes creating the shape tables a trivial exercise.

SHLOAD. A command to load a shape table from tape.

SOFT SWITCH. A memory location which controls something. You may set and interchange output modes using soft switches.

TWO'S COMPLEMENT. A binary number formed from another by interchanging ones and zeros and then adding one.

USR. A BASIC command which passes a value to a machine language routine. The starting address for the routine must be placed in locations $ØB and $ØC, while a $4C must be put into $ØA. When the routine is finished, control returns to the BASIC program.

VECTOR. In shapes, a pair of instructions which tell the computer whether to plot at the current position, and which way to move next. For my math professor, an element of a vector space.

VECTOR DIAGRAM. A figure drawn on paper showing all of the vectors as arrows.

VLIN. A BASIC command used to draw a vertical line in Low-Res.

XDRAW. An APPLESOFT command similar to DRAW. XDRAW switches on any pixel of the shape which is off, and switches off any pixel which is on. XDRAW is very useful for erasing shapes from the screen.

Index

Addition, 168-172
Address(es), negative, 173-174
Addressing, 6-7
ALTCHARSET, 36, 37
Animation, 135
  byte-move, *see* Byte-move process
  flicker in, 155
APPLE EXC., 57, 58
Apple screen characters, 175t
APPLE TEXT, use, 112, 113
APPLESOFT extensions, 21-30
ASCII character codes, 32, 176t

Bar graph, creating of, 119-123
BASIC, memory move, 59f-60f
Binary, 3, 163
  adding and subtracting, 170-172
  conversion of decimal to, 166-167
  conversion to decimal, 166
  difficulties in, 7
Binary/hex chart, 8f, 9
Binary/hex examples, 9, 9f
Binary number
  conversion of decimal to, 168
  writing of, 9
Binary nibble, 9
Bit(s), 4, 5, 171
  color, 82-84, 92, 96
  combinations, 5f, 5, 6
  Hi-Res, 68, 69
    dot correspondence to, 69f
  patterns, 6
  value of, 5-6
BLOAD command, 51, 116
Blocks, plotting and erasing, 48-49
BSAVE command, use, 51, 53, 106, 107
Byte(s), 6, 171
  address of, 6-7
  divisions, for vector encoding, 102f, 102, 103-104, 105
  value of, 8, 9, 162
    changing of, 14, 15
Byte-move process, 135-136, 150
  animation by, 136-138
    altering of, 138-139
    for horizontal animation, 139-142
    two-byte animation, 142-145
Byte-move shapes, pre-shifting with, 154

Calculation, 149-151
CALL command, use, 21, 24-26
  in memory move, 60
Character codes
  Apple screen, 175t
  ASCII, 176t
Circle graphs, 129-132
Circuits, 4
Clashing, 95-96
80 COL, 36
Collision, detecting of, 157, 158-164
Colon, use, 15
Color
  Hi-Res, 81-96
  selection of, 123
Color bit, 92, 96
  for Hi-Res graphics, 82-84f
Color codes, Low-Res, 44f
Command(s), 2, 19
Command extensions, 21
Complements, forming of, 172-173
Computer, physiology, 3-11
CTRL-C, 6, 27, 159
Cursor
  movement, 50, 50f
  positioning of, 123

Data, 55
  displaying of, 119
Data file, 55-57
Data processing, 119
DATA statements, 119, 137
Decimal, 3
  conversion of, to hex, 167-168
  conversion to, 9, 10
    of binary, 166-167, 168
    of numbers, 165-166
Digit(s), position within numbers, 165
Digital, digressing to, 4
Digitizing, 99, 100, 109, 116
Dithering, 81, 93, 94
3D0G command, 19
DOS, 15, 16, 18
Dot(s)
  Hi-Res, 65, 67, 68, 69-78
    double dot picture, 92, 92f
    quantity available, 91-93
  isolating of, 162
  patterns, POKEing of, 149
  plotting of, 109-110
  processing of, 90
  shape plotting and, 100, 101, 103
Draw, use of, 105-106
DRAW command, 108, 113, 114, 116, 135

Editor(s)
  Hi-Res, 66, 100
  Low-Res, 48, 49-50
EOR operation, 154, 174
Equipment, 1
EXEC, use, 57-58

Figure(s)
  digitizing of, 99, 100
  saving of, 50-51
Filename, 56, 57
FOR-NEXT loop, 54, 128
Functional plotting, 128-129

Glossary, 181-185
GOTO statement, 28, 29, 158
Graph, graphing
  circle graphs, 129-132
  drawing, 128-129
Graphics, speed and efficiency, 147
Graphics mode, 31-33, 35, 35f
Graphics switch, setting of, 34

HCOLOR, 92, 106, 116
Hex, see Hexadecimal system
Hexadecimal digit, conversion to number, 166
Hexadecimal number
  conversion to decimal, 165
  writing of, 9
Hexadecimal system, 3, 7-8
  adding and subtracting, 168-170
  binary/hex chart, 8, 9f
  binary/hex examples, 9, 9f
  conversion of decimal to, 167-168
  conversion to decimal, 9-10
  digits, 9
Hi-Res color, 91, 96
  clarification, 90-91
  color bit, 82-84
  colors available, 93-96
  dots available, 91-93
  positions possible, 91
  program, 84-90
Hi-Res graphics, 62, 106
  display, 68
  dot pattern, 65, 69-78
  memory, 66-68, 161
  use, 65
Hi-Res graphics editor, 66, 100
Hi-Res memory map, 69, 70f
Hi-Res mode, 31, 32-33, 35f
Hi-Res screen, 111
  clearing of, 73
  dots available, 91-93
  positions available, 90
  text placement on, 112-113
  use, 164
HIMEN, 59, 60, 168, 169
  setting of, 107, 108
HLIN command, use, 21, 46, 47f, 47
Horizontal animation, 140-142
Horizontal axis, labeling of, 128
Horizontal graph, 121
Horizontal pre-shift, 152

Line graph, drawing of, 123-128
Loop, checking of bits, 162
Low-Res color codes, 44f
Low-Res editor, 48, 49-50
Low-Res graphics, 62
  creating pictures, 44-46
  use, 43-44
  memory, 53, 54, 58
Low-Res memory map, 41f
Low-Res mode, 31, 32, 35f, 39
Low-Res scan, 54f, 55, 55f
Low-Res screen
  memory, 58
  placing text labels on, 121

Memory, 4, 13, 161
  changing of, 15
  dealing with, 48-50
  examining of, 14-15
  Hi-Res, 66-68, 161
  interpretation of data, 31
  Low-Res, 53, 54, 58
  moving of, 17-18, 58-61
  organization of, 15-16
  scanning of, 53-55
  size, 7
  writing from, 16-17
Memory address, 7
Memory map, 16, 16f, 18, 177f-179f
  Hi-Res, 69, 80f
  Low-Res, 41f
  use, 40-41, 41f
Memory move, BASIC, 59f-60f
Modification, partial, 147-149
Monitor, 13-14, 15
  commands, 19
  use, 14-15, 73
MOVE command, use, 17, 18, 59

Negative address(es), 173-174
Nibble, 9
Number, digit position within, 165
"Number crunching," 119
Number table, 123, 124
Numeric codes, assigning of, 32

Offset, 105

"P" key, 27, 50
Paddle 0, use, 154-155, 159
PEEK function, 34, 37

PEEK statement, use, 13, 21-23, 59, 161
Pictures
  Low-Res, 44-46
  preserving of, 53-63
Plot, hatching of, 46-48
PLOT command, use, 21, 46, 47, 48
Plotting, functional, 128-129
POKE command, use, 13, 21, 23-24, 25, 99, 108
POKE function, 34, 37, 99, 138, 149, 151
Power light, 4, 6
Pre-calculation, 149-151
Pre-shifting, 141-154
PRINT PEEK command, 22, 59
"PRINT USR(C)" statement, 27-28
Prompt characters, 14

RAM, 4, 5
  altering of bytes, 14, 15
  erasure from, 5
READ command, 51, 99, 108
ROM, 4, 5, 13, 14
ROT, use, 108-109, 110

SCALE, setting of, 106, 108
Scaling factor (SCL), 122, 127
Scan, Low-Res, 54, 55
SCL, see Scaling factor
Screen
  clearing of, 16, 17
  shape generation on, 101
SHAPE, use, 99
Shape(s)
  byte-move, 154
  creating and placing of, 100, 109-115
  designing and digitizing of, 100, 116
  drawing of, 100-103, 105
  finding of, 105
  plotting of, 99
  UFO's, 103-104
  use of DRAW, 105-106
  use of ROT, 108-109, 110
  use of XDRAW, 115-116
Shape commands, 135
Shape table, 104-105, 111, 123, 127, 158, 168
  entering and saving, 106-108, 123-124
  location of, 105-106, 111
  shape drawing with, 109, 110
Shifting, 151-154
SHLOAD command, use, 108, 116
Soft switches, 33-34
  addresses, 35f
  for display control, 36-37
  toggling, 33, 34
Sound assembly, 61-62
Subtraction, 168-172

Tape, saving of, 107-108
Text, placing of, 112-113, 121
Text modes, 31-32, 35f
  memory map, 40-41
  use, 39-40, 41-42, 43
Text screens, 31, 32
Toggle switches, 33, 33f
Trigonometry, circle drawing with, 131

UFO's, drawing of, 103-104
USR statement, use, 26-29, 31

Vector(s), 135
  digitizing of, 109
  shape drawing with, 100, 103, 135
    encoding of, 101-103
Vector codes, 101f
  placing of, 102
Vertical address, 42
Vertical blanking, 155
Vertical Blanking Location (VBL), 155
VLIN command, use, 46, 47f, 47, 48
Voltage, testing for, 4

XDRAW command, 135, 152, 153, 154, 155, 159
  use, 115-116

"This book clearly explains a difficult and complex area with understanding, style, and wit. . ."
—Theodore Klein, Boston Systems Group

"Almost indispensable for anyone who wants to create graphics on an Apple II computer. . ."
—Robert Fisher, Computer Science, De Paul University

Includes Graphics Programming Features Available On The Apple IIe!

Apple II Computer Graphics

Ken Williams, Bob Kernaghan, Lisa Kernaghan

Finally—a clear, concise, state-of-the-art treatment of the graphics capabilities for the Apple IIe—designed for users with a working knowledge of BASIC programming. Here you will find the complete range of the most current graphics techniques, from placing dots on the graphics screen to artificial color generation, animation, and even sophisticated Byte-Move techniques. Written in an easy-to-read, self-study style, it's the only book you need to create spectacular high quality computer graphics without assembler or machine language skills.

In this book you will find:

- A complete explanation of the Apple II and IIe graphics
- Techniques for programming and designing many of today's popular computer games
- A complete section on business and technical graphics
- Stimulating end-of-chapter exercises
- Examples in assembler language for advanced users
- A complete glossary and continual cross-referencing throughout!

CONTENTS

Introduction / Computer Physiology / System Monitor—Memory Tricks / APPLESOFT Extensions / Graphics Modes and Soft Switches / Text in Low-Res / Preserving Your Pictures / Hi-Res Graphics / Hi-Res Color / Shaping Up / Graphs and Charts / Byte-Move Shapes / Advanced Moves / Collision Course / Decimal, Hex, and Binary / Appendices / Glossary / Index

ISBN 0-89303-315-4